Engaging Weak Programmers in Problem Solving

Brad Alexander and Cruz Izu
School of Computer Science
The University of Adelaide
Australia
{brad,cruz}at cs.adelaide.edu.au

Abstract— **As with many schools attracting international students, our postgraduate degrees must cater for students with diverse backgrounds and skill-levels. It is not practical to accurately assess students' domain-skills prior to enrollment. Thus, we found a need for a compulsory bridging course with the dual objectives of improving the problem solving and programming skills of the weakest computer graduates whilst still challenging and improving the skills of experienced programmers. This paper describes our experience in refining such a course over six semesters, in order to meet these goals.**

Keywords: collaborative learning, programming, problem solving, self-reflection

I. INTRODUCTION

Like many computing schools exposed to the international market our school attracts students from a diversity of backgrounds and skill levels. The diversity of skills is most apparent in our postgraduate coursework students with many having good theoretical knowledge but lacking in practical programming skills. To address this weakness, we developed a compulsory course (named Specialised Programming) to be taken by all starting Masters students. The main challenge for this course was to help weaker programmers to reach a satisfactory level while also challenging and improving the skills of the more experienced programmers.

Inspired by ideas from Astrachan[1] in first year undergraduate teaching, we based our course on practical application of programming to small, but realistic, TopCoder[2][1] problems. To cater for our cohort, we modified his approach in two key aspects:

- the use of cooperative learning strategies[3] to support the transfer of skills between students.
- course assessment based *primarily* on graded sets of these realistic problems, to better assess and directly address specific deficiencies in problem-solving skill sets.

Although other courses use subsets of TopCoder problems to replace/extend coding assignments, our course aims to improve the programming skills of postgraduates by intensive practice at solving these problems. The design of this course is informed by principles of deliberate practice[4,5]: skill acquisition is based on high repetition, designed practice, with fast feedback, tailored to individual abilities. The requirement for fast feedback means that errors should be identified and corrected during practice. The acquisition of expert performance through deliberate practice typically takes years [5, 6] so we cannot produce expert programmers in a 12-week teaching semester. However, we should expect to see a measurable improvement in student programming skills, which may provide self-motivation for further practice.

This article describes our experience with presenting this course over six semesters and in particular the impact of a number of changes made at the start of 2009 based on the experience of students with weaker programming skills.

The remainder of this article is structured as follows. Section II describes the course structure followed by the results and insights made during its delivery in 2007 and 2008 in Section III. Section IV lists the changes proposed at the start of 2009 in order to engage weaker programmers. Section V shows the impact that changes made on performance, particularly on the bottom half of the class., followed by a summary of what we have learned in the process.

II. COURSE STRUCTURE

The course presentation consists of lectures, structured group practice sessions, and ad-hoc tutorial sessions as needed. Except where stated, the following description of course structure applies to all six semesters that have been presented so far. Refinements to this framework over time are defined in subsequent sections.

Lectures are run in two-hour weekly sessions presenting algorithmic techniques for various problem types, based on the textbook "The design and Analysis of Algorithms" by Anany Levitin. The end of each session is typically devoted to supervised programming practice.

Three hours of each week are devoted to either practical exams or structured group practice sessions. The group practice sessions offer a mix of new and familiar problem types. Each group, of approximately four students, is assigned one or more problems. After working on the problems for 60 to 90 minutes, a group representative is picked to present their algorithmic solutions to the rest of the class. Coding and submission is to be completed independently by each student after the practical session. A summary of the problems selected for each session is provided at [7].

Each TopCoder problem statement has a consistent style, starting with a precise and entertaining description of the problem; followed by a description of the class name and

[1] Questions used, with permission, from TopCoder Inc.

method signature required; input data constraints; and, finally, some sample input and output data. Hints are rarely given in problem statements and there are usually many good ways to answer each problem allowing room for creativity.

Testing sets for each problem are harvested from TopCoder data using scripts and converted into a Java driver program to be used in the School's automatic submission system. During submission, students get feedback on the number of test cases they passed and for the first test case they fail they are given the input and expected output. They can resubmit their code as often as needed.

Groups are initially allocated in week 2, based on the result of a diagnostic practical exam using a genetic-grouping algorithm, to maintain diversity of skill levels within groups and uniformity between groups. Group allocations may be modified based on the performance up until the mid semester break.

Students are expected to practice outside of class time. They are encouraged to practice problem-solving with their group, but code their solutions individually. Group work makes the experience less threatening and decreased isolation.

Assessment is primarily based on practical exam performance with smaller amounts of marks allocated to group participation. Assessment begins with a diagnostic exam early in the semester followed by practical exams every 2-3 weeks. The minimum requirement for a passing grade will be to solve at least one problem in each exam. The submission system remains active at all times and students are able to practice submitting any question from the practice sessions or previous exam sessions as part of their practice regime.

Practical exams are hosted in a reasonably secure online environment. During the practical exams, students' normal accounts are suspended and practical exam accounts activated. Network controls are put in place to restrict access in the laboratory only to the school web pages, the submission system, and individual subversion accounts, in which the students place answers to be harvested by the submission system.

Each practical exam consists of three problems of differing degrees of difficulty. The first problem would require simple algorithm techniques (i.e. sorting, searching, string manipulation) and basic knowledge of the java API (documentation for which is available on-line during the exam). The other two problems will use the techniques taught during lectures (i.e. brute force, greedy, recursion, dynamic programming). The exam time is set to 3 hours, including the reading time (compared to 75 minutes for a Top Coder competition), so there is time to correct and modify the solutions, although a student needs to be a good programmer to solve all three problems in that time.

III. INITIAL RESULTS

In this section we describe the outcomes of the first four offerings of the course in 2007 and 2008 in terms of three areas: assessed results; the quantity and quality of student practice; and student performance in exams. We describe each of these in turn.

A. Assessed Results

All four initial offerings enjoyed reasonable success with almost all students being able to solve at least the simple problems and achieve a passing grade.

As an indication of the initial ability of each cohort, Table I shows the performance of students in the diagnostic exam for the six consecutive semesters it has been taught. The problems in the diagnostic exam cover a range of the basic problem solving schemes and we would expect a graduate student with average Java programming skills to solve two problems. In all intakes, there was wide range of programming skills.

TABLE I. STUDENT PERFORMANCE ON THE DIAGNOSTIC EXAM

Problems Solved	2007		2008		2009	
	S1	S2	S1	S2	S1	S2
0	17%	18%	42%	22%	71%	40%
1	8%	12%	33%	11%	12%	30%
2	25%	18%	17%	41%	12%	20%
3	50%	53%	8%	26%	6%	10%
	original course				after changes	

The first cohort of the course and, to a lesser extent, the second cohort contained a number of students who were reasonably competent programmers with enthusiasm for problem-solving. These offerings worked well for these students, with several commenting positively on the opportunity to be challenged and learn new skills. The ability of these cohorts was, in part, reflected by a high number of credit and distinction grades in the first two semesters.

In 2008 the average skill of the cohort dropped significantly. This can be attributed to the fact that, 37% of students who enrolled in Semester 1, 2008 had deferred enrolment in that course during 2007. Similarly, the Semester 2, 2008 intake had 27 student of which 8 have already deferred enrolment at least once. In other words, the weakest programmers delayed taking this course until the end of their programs (the program rules made the course compulsory but did not enforce pre-requisites, so it could be taken any semester). The data for 2009 is included in Table I as a basis for later comparison.

Students initially lacking programming skills could solve few problems during their diagnostic exam and struggled to both solve and code simple problems in practical exams. The apparent causes of these problems varied. Some have issues understanding the problem descriptions but with help they were able to code the solutions on their own. Others had little experience with Java and needed help to process any non-trivial inputs values. Others had difficulty debugging. To address these problems we provided ad-hoc tutorial support for those who felt they needed it.

Student skill levels in each semester improved from the diagnostic, with low failure rates below 10% in all semesters

(without including, as we will discuss later, those who failed due to cheating in 2008). Although some students showed low performance in the first two practical exams, as they improved on later exams they were offered a supplementary exam.

B. *Student Practice*

Practice is the core activity in this course and students were encouraged to practice individually and in groups on a broad range of problems. Over 5000 individual practice submission attempts have been made by the 93 students who have taken this course over the four semesters Practice problems, of graduated difficultly are set up in the same way as the exam problems. Students can access practice problems on the web and submit multiple times without penalty. In addition, previous exam problems remain online as an additional pool of practice problems.

Due to the limited time in supervised practical sessions, the last step of problem solving - the coding of the algorithm - is left for the students to do outside the class at their own pace. Unsurprisingly, the volume of practice undertaken outside of class varied considerably between students. Fig. 1 plots, for each student in each semester, the number of practice attempts in the automatic submission system versus the number of attempts achieving a 100% score.

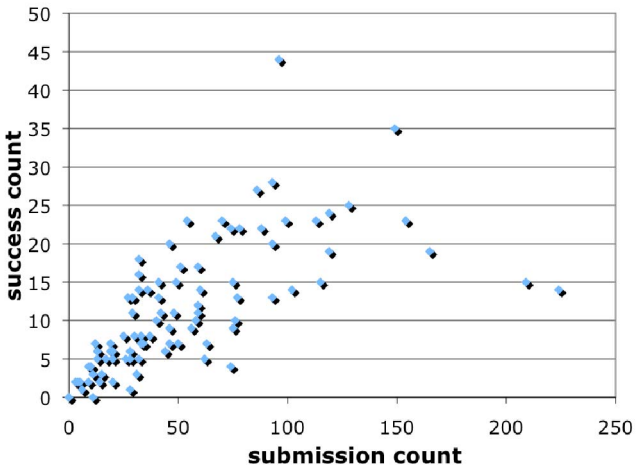

Figure 1. Number of practice problem attempts versus number of practice problems solved (2007-2008)

The plot shows that most students make about four total attempts for each successful attempt. This is not unexpected, even seasoned programmers sometimes fail to anticipate some test cases or make minor errors in their code requiring multiple submissions. The plot also shows the number of attempts made and the number of problems solved is highly correlated (correlation coefficient = 0.67) indicating that in most cases intensive practice is rewarded with a reasonable amount of success (or, conversely, that success encourages more practice). Last, but importantly, the plot shows that most students make less than 50 practice submissions solving less than ten problems – well below our recommended practice to students to solve 1 or 2 problem questions each week in a 12-week teaching semester.

Another observation about practice is that students did not practice consistently throughout the semester. Fig. 2 illustrates this issue in the 2008 semester 2 instance of the course. We observed students doing practice problems in the first few weeks leading to the first practical exam at the end of teaching week 4, but after that the amount of practice declines to very low levels in the second half of the course. This is in spite of the fact that that 3 of the 5 practical exams, which might benefit from practice, are held later in the semester.

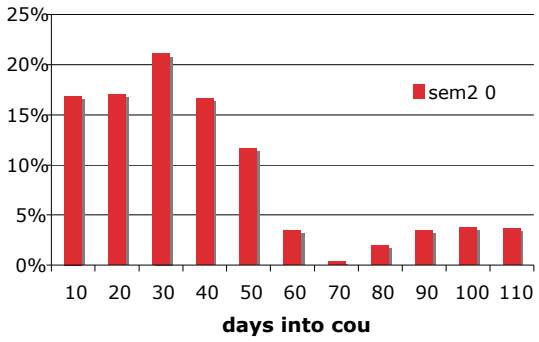

Figure 2. Histogram of practice submissions over duration of the semester 2 2008 course.

Motivation for practice also varied considerably between students. This variance was anticipated and, in the original design of the course, group members shared a proportion of their exam marks to create some of the positive interdependence needed for cooperative learning [8] and reduce the propensity for social-loafing [9]. Unfortunately, in the first instance of the course, a significant proportion of students - aiming for a bare pass - optimized their level of practice to the detriment of other group members. So, in the interest of fairness, the sharing of marks was quickly withdrawn under a no-disadvantage rule.

So far, we have seen that the amount of practice and the motivation for practice varies between students and across time. What about the impact of this practice on exam performance? As might be expected, the number of practice problems solved, by individuals is correlated with the number of exam problems solved by those individuals. Fig. 3 plots these metrics for each student in each semester. The maximum number of exam problems it is possible to solve in a semester is 15 (three problems in each of five practical exams).

The correlation between practice problems solved and exam problems solved (correlation coefficient = 0.34) is weaker than the relationship between practice submissions and practice success. This weaker relationships might be attributed to a number of factors: the time limit in the exam gives students more limited scope for experimentation; exam conditions restrict access to resources that may help with problem completion; some experienced students didn't practice and simply turned up to the exams; some students aimed for a bare pass; and, finally, a minority cheated.

These *in-exam* performance factors bear closer scrutiny, thus next section will explore some of these aspects.

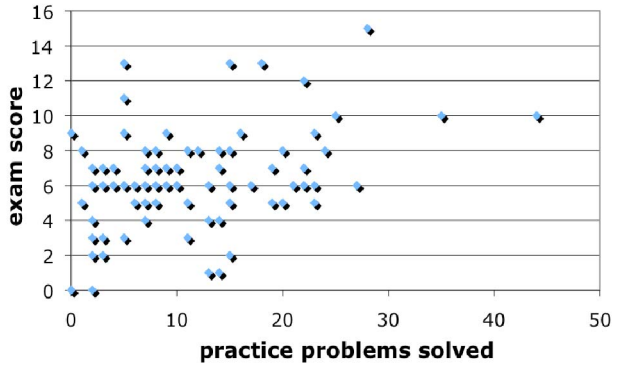

Figure 3. Number of practice problem solvedversus number of exam problems solved (2007-2008)

C. *Student Exam Performance*

In this section we describe aspects of student behaviour and performance during practical exams. In general, students are more prone to rush in the exams than during regular practice until they have solved the first problem, which keeps them in the pass range.

When submitting a coded solution, the student receives immediate feedback on the number of test cases passed, and when incorrect it can see the first test case failed and the expected output. For some students this feedback is sufficient to refine/correct their code, but weaker programmers spend most exam time fixing failed input cases one by one, breaking working parts of the code in the process. Their final solutions are longer that needed, and as the problems get harder, they cannot produce a working solution for all cases.

We present here as an example the analysis of the submitted solution for the problem *BoxesofBook*s, during a practical exam in semester 2, 2008. A solution to this problem requires a simple loop to iterate over the array of books to be packed into boxes, returning the number of boxed needed, and the overall code should take 20-25 lines at most (reducible to 15 lines if you don't use nice layouts). Good coders solved the problem in a short time interval, within 3 attempts or less and their code size was as expected. Other coders took longer and although they solved the problem their final solutions were complicated and their code size went up to 30-45 lines. As expected, later submissions increased code size; in only *one* case a submission reduced the code size, after the student worked out he was going in the wrong direction.

Four students (out of a class of 27) were not able to solve the *BoxesofBook*s problem. Two of them started with a reasonable solution but they keep adding code to solve each case and ended up with 50 and 82 lines respectively. One of the students submitted 23 attempts, with only a few minutes between them. Another student keep changing conditions within the loop but could not correct their code after 13 attempts.

The extent of this problem of repeated attempts is apparent in Fig. 4, which plots the number of exam submissions made by each student in each semester against the number of successful exam submissions for that student. The plot shows a worryingly low success rate for some students with, in one case, over 166 submissions for 3 questions solved.

The anatomy of successful and failing exam attempts is analysed in Fig. 5, which contrasts the pattern of attempts for exam problems leading to success (881 successful attempts) with those leading to failure (423 eventual failed attempts). As can be seen, most successful attempts at questions take three or less submissions to get it right. Failed attempts tail off considerably more slowly. Worryingly, more than 5% of failed attempts have more than 15 submissions. These high counts almost always indicate that a student is re-submitting by making minor unproductive changes in a desperate attempt to pass the automatic testing. We label this unproductive behaviour in practical exams as thrashing.

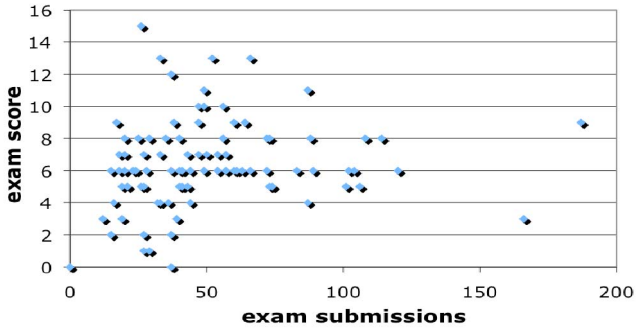

Figure 4. Plot of number of exam submissions made versus number of exam problems solved (2007-2008).

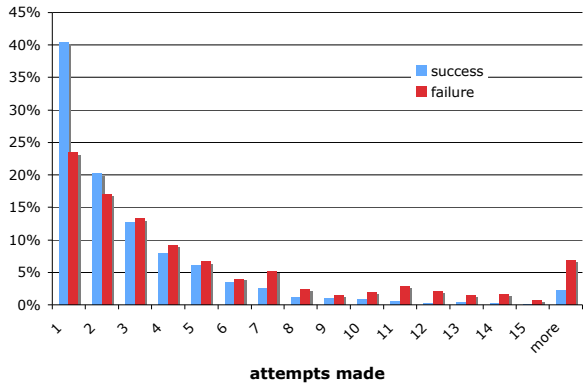

Figure 5. Histogram of submission patterns for successful and failed attempts at exam questions.

The impact of thrashing is significant. Given a conservative threshold for thrashing of three or more unproductive attempts in less than 15 minutes, 30% of the nearly 6000 submissions made in practical exams are instances of thrashing. As an aside, not every instance of multiple closely spaced attempts at a question constitute thrashing.

A small number of, eventually, successful attempts take a very large number of submissions. Sometimes a large number of submissions stems from an attempt to pass the automatic

testing by the tactic of: (1) failing a case (2) harvesting the expected output value (3) inserting a print statement of the expected output value and (4) then resubmitting to harvest the next value and so on. Given that the number of test cases is usually high, in the range 60-120, the effort required for this tactic is considerable[2].

Because so much thrashing was occurring in exams and, to a lesser extent, during practice we decided to investigate this pattern more closely. In some practical exams we observed the weaker students attempted the first problem and after a few attempts to fix their code they gave up and move to the next problem. Again, they were able to produce a draft solution that passed the trivial cases but were not able to fix their code in order to pass the non-trivial cases. As the end of the exam approached, they panicked and jumped from one problem back to the other one, editing their code but their corrections were not well thought through. For example, they would panic and reverse the condition of a correct *if* statement to see if that passed more tests.

It is not surprising, given the pressure to complete their masters program, that some students resorted to cheating during practical exams. In Semester 1 2008, 10% of students failed due to documented cheating, although there was other doubtful cases which avoided being failed, due to the complexity of detecting and proving cheating as discussed in [10]. Although we have removed the submissions identified in any cheating incidents before analyzing the data, there is some doubt on the real performance of some students under different exam conditions for semester 1, 2008. Simple measures were taken to make cheating more difficult and the rates of suspected cheating have fallen in 2009.

Finally, two common behaviours for less-skilled programmers were a passive role in their group practice and diminishing attendance lectures as the material moved on to address more advanced concepts.

D. Insights 2008 intake

At the end of 2008, we reflected on the performance of the struggling students compared with those who thrived and gained the following insight:

All students were able to sketch a feasible solution for most types of simple problems. But weak programmers often overlooked one or two boundary cases and, thus, they failed some test and they were not able to identify the causes of their failure.

We will list below the list of factors that prevent weaker students from solving non-trivial problem:

- Lack of debugging skills: although students do use debugging tools and print statements to find what their code is doing, they are slow in both indentifying the line(s) of code that are causing the problem and the changes needed to fix it. This is general caused by limited programming practice.
- Poor coding style: they code monolithic solutions, and rarely define additional methods to simplify coding and testing. For example, given a string manipulation problem in which they need to identify vowels, they would not define a method called *isVowel(*char x), instead they repeated the code needed to detect this condition as many times as needed.
- When their code fails after several changes, students are reluctant to go back and work through their solution on paper or change their strategy, as they felt this would take too long.

[2] And for naught - these cases are manually detected and given zero marks.

IV. CHANGES TO ENGAGE WEAKER PROGRAMMERS

This section will report the changes to the course delivery aimed at keep less-skilled programmers engaged, by addressing the set of problems identified above.

A. Introduction to problem-solving

From the analysis done, it was clear the course pace was too steep for those with low programming and problem solving skills.

The textbook assumed good programming skills and skipped over the range of problems that need simple algorithmic skills (sorting, searching, string manipulation, simple maths). Therefore, there was no support for the weaker student that lacked basic problem-solving skills. In other cases, students were not familiar with the Java API so they wrote their own code instead of using simple methods (such as *Maths.min* or *Array.sort*) to process the data.

So, the first change was to extend the coverage of basic problems types from one to three lectures. The additional time allowed to provide more examples and to describe ways to identify each basic problem type. Through the examples, we emphasized the need to reference to the Java API to find methods that may simplify coding. For example, when describing a string manipulation problem that uses prefixes we asked students to go through the String class and find methods that may be useful (such as *startsWith).*

B. Step-by step problem-solving

Weaker students write their code without prior testing of their strategy and made limited use of the given examples provided to see if their code will work.

To emphasize the need to devise a basic algorithm, we introduced a paper problem template, with sections to identify the problem type, write the basic idea of how to solve it, and then convert it into an algorithm. This template was provided at each lecture time, and we asked students to work on the template before coding the solution on the terminal. Instead of moving to the labs to do a problem-solving session at the end of each lecture, we ran them in the lecture room, forcing them to write their solutions on the template. We also introduced in the first practical session techniques to test the examples on paper, so we can see if the algorithm works for the examples given before coding it. The additional teaching materials and modified lecture schedule are provided in [7] for the interested reader.

These changes, combined with some one-on-one coaching, helped most students to focus on doing some development and testing of algorithms prior to coding.

C. Increase group practice and feedback

Another major goal to keep weaker programmers engaged was to enforce problem-solving practice every week, particularly on the second half of the semester. To achieve this, we ask each member of a group to fill an online report weekly, from week 4 onwards, describing the list of problems attempted and/or solved and the way the group tackled them. The report format is shown in Fig. 6. Each weekly report was worth 1% of the final mark (9% in total).

Although the reports are not compulsory, each student must complete at least 40% of each of three course components (1.practical exams, 2.practical sessions and 3,group reports) to avoid a failing grade.

By asking them to list the Java classes and methods used to solve problems, we reinforce each week the need to check and use the Java API. Thus, there is a minimum practice required outside the practical exam, which is monitored by the weekly reports. The second question provides feedback to lecturers on group dynamics, and by having to report on it, we encourage weaker students to participate in their groups. The last question encourages self-reflection so that each student could identify and assess the strategies they were using.

1. List the problems you attempted, your scores so far for each problem, and which Java API Classes and methods you used to solve each problem. Remember that you must attempt at least two different questions each week. For example:

SlowKeyBoard: 90 out of 90
- ArrayList (size, add, get)
- Arrays.sort
.
FracCount: 7 out 100
- HashMap (containsKey, put, get)
- ArrayList (size, add, get, remove)

2. Describe what happened during your group practice session. Who contributed and how did they help the group? Highlight the things that went well and/or any problems that arose.

3. Which steps of the problem solving process did you find easy?
Which steps did you struggle with?
What approaches did you use to overcome those difficulties?
Can you think of how you might approach the problem differently if you had to start again?

Figure 6. Weekly Group Report online questions.

We are able to provide feedback to their self-reflection when marking the reports. This refinement and review process is aligned with other methodologies for addressing problem solving in Computer Science [11].

In the first semester 2009 we focused on feedback and self-reflection, but noticed some students were claiming to complete problems they did not have submissions for. To counter this, in the second semester we policed their submissions and sometimes reminded them of the need to code their own solutions.

The weekly reports were also useful for the lecturers by providing feedback during the course on the topics and issues than students find more difficult so that teaching materials could be adjusted accordingly.

V. RESULTS

The new course structure has been used in both semesters on 2009. In this section we will evaluate the impact that the changes had both on regular practice and student performance.

There was some anecdotal evidence from two students repeating the course in Semester 1, 2009 that the gentle introduction to problem solving made the course content less scary; the examples provided in lectures were a good start to review the java API, and students felt more confident before taking their first practical exam in week 4.

We should note 2008 had the largest cohorts (36 and 27 students for semester 1 and 2 respectively) while 2009 had small cohorts (19 and 10 students). Particularly, students in the last cohort had nowhere to hide, they received more feedback and the chances of cheating being undetected were much reduced. In fact, two students cheated in the first practical exam, but as they exam results were so different from their class performance, they were easily identified and they decided to drop the course, so they are not included in the study.

A. Problem solving practice

The changes of the course encourage students to keep practicing during the whole semester. Groups meet outside the supervised practical session to solve problems and the weekly reports provide feedback of the interaction in the group. In the second semester, groups were more active. Although there where some issues with students who did not participate, 80% of the student agreed that working in a group was a valuable experience.

Fig. 7 shows number of practice problem submissions per student per term in 2009 versus the number of problems actually solved. The old 2007/2008 data from Fig. 1 is included for reference. Success rates in 2009 are similar to 2007/2008 but the average amount of practice per student is slightly higher and the correlation between practice attempts and practice success is higher (0.90, up from 0.67) this is partially due to there being fewer students getting little reward from their practice attempts.

This better success rate also appears to be reflected in exams. Fig. 8 plots the number of exam submissions per student per term in 2009 versus the number of problems solved

in exams. The old 2007/2008 data from Fig. 4 is included for reference.

The plot shows that there are comparatively few students making many submissions with limited success. This is also borne out in the higher correlation between exam submissions and exam success in 2009 (0.76) than in 2007/2009 (0.51). Overall, the amount of thrashing in exams was reduced.

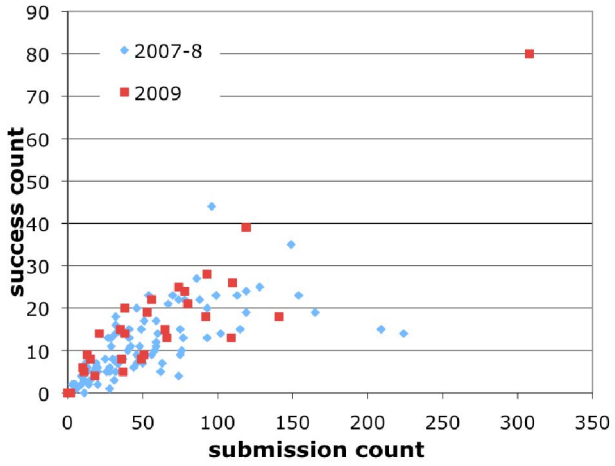

Figure 7. Number of practice problems attempted versus the number of practice problems solved in 2009 (2007-2008) data included for reference.

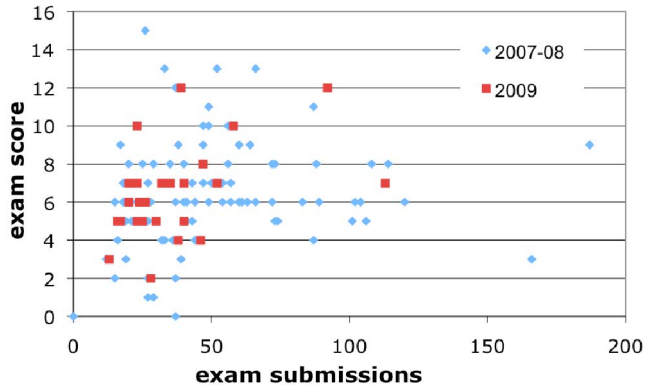

Figure 8. Number of exam submissions made versus the number of exam problems solved in 2009 (2007-2008) data included for reference.

Practice habits also improved somewhat in 2009. Fig. 9 shows the amount of practice over the semester for semesters 1 and 2 of 2009 with the data for semester 2 of 2008 from Fig 2. included for reference. Some improvement in second half practice is apparent.

Table II shows the number of problems solved both during practice and during the exam for all students who remain enrolled for the whole semester. We should note that we can only measure practice done using our submission system, which may be lower than expected if students practiced at the TopCoder website. Our local problem pool had approximately 40 problems in 2007, and grew by 15 problems each semester, so in the latest semester there are more than a 100 problems to choose from. Only two out of 10 students in the last semester reported using TopCoder to source additional problems.

The rise on problem-solving practice is more pronounced in semester 2, as we reminded students on the need to code their solutions via the report feedback. Consequently, the number of problems solved raised to 20 - within course expectations as they must submit 9 weekly reports, in each of them they should report on solving two problems. As the problem sets in the practical sessions become harder, some students choose to solve problems from previous year's practical exams[3], which is in line with deliberate practice.

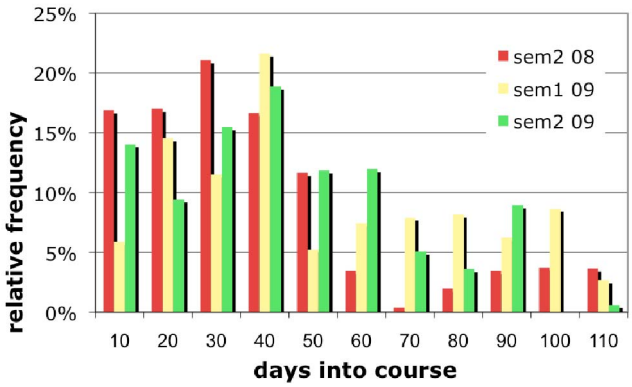

Figure 9. New practice patterns in 2009 as compared to semester 2 2008.

TABLE II. AVERAGE PER-PERSON SCORES

Average per-person	2007		2008		2009	
	S1	S2	S1	S2	S1	S2
Practice attempts	80	41	53	48	55	77
Practice solved	11	9	13	12	14	20
Exam Attempts	54	38	58	44	37	32
Exam solved	6	5	8	6	6	7
Course Score	71	62	63	51	65	67
	original course				after changes	

B. *Exam performance*

Row four of table II shows the number of problems solved during practical exams. The number of problems solved is a good indicator of performance, although the higher value in 2008 S1 is a possible reflection of the cheating problems that were detected in the middle of the course.

We can see the number of exam attempts is lower in 2009 compared to 2008, this is an indication of reduced thrashing.

[3] In all our analyses we count these attempts at old exams as practice attempts rather than exam attempts.

It is difficult to measure class coding speed as each problem set may be more or less challenging depending on the type of problem chosen and the Java skill-set required. Figure 10 compares the speed to solve the first problem for three consecutive cohorts, one before the changes to the course, and two after. To reflect the problem complexity we chose the second fastest solution in each exam as representative of the top of the class and compared that time with the time needed by the bottom half of the class. We measured the speed of the bottom half of the class using two metrics:

- *half-solved* is the average time taken by those in the bottom half of the class who manage to solve the problem
- *half-all* consider that those who did not complete will take at least 3 hours to do so, and include them in the average.

Thus, a higher half-all (the green bar) compared to the half-solved (red bar) indicates some students failed to pass that exam. We can see this is the case in all exams in semester 2, 2008, except for exam 3.

The exam performance for the bottom half of the class was, in general, better in both semesters of 2009. Students in semester 1 did not show significant improvement compared to the previous cohort, but considering their initial low skills (please, refer to Table I), there was a clear improvement as the semester progressed. At the end of the semester all students were confident they could solve at least one problem, which may have also helped to keep them calm and do better.

Although exam difficulty is variable, we can see that the difference in speed between the best coders and the bottom of the class is narrowed in the last semester. The best performance in 2008-S2 was for exam3, problem 1, when all but one student solved the first problem. The question was simple and it took 13 minutes for the second fastest coder, and the slowest one took 2 hours and 46 minutes. A similar type question was used in 2009-S2 exam 5, and as the figure shows, it also took 13 minutes for the second fastest coder, but the average for the bottom half was 28 minutes and the slowest student took only 37 minutes.

We believe there were two reasons why the last cohort solved some problems faster:

1. they were able to identify the problem type (brute force) as they read the problem description,
2. they needed less time to debug/correct their code.

We should note than in exams 3 and 5 of semester 2, 2009, all students managed to complete the first and second questions, compared with exam 2 in which 3 students could not solve one problem. Exam 4 had a harder first problem but most students were able to complete it.

There is also evidence in the practical sessions that students became better at problem-solving, and could identify the problem type and some basic algorithms to be applied. However, their problem-solving skills were ahead of their Java programming skills so they could not code their solutions.

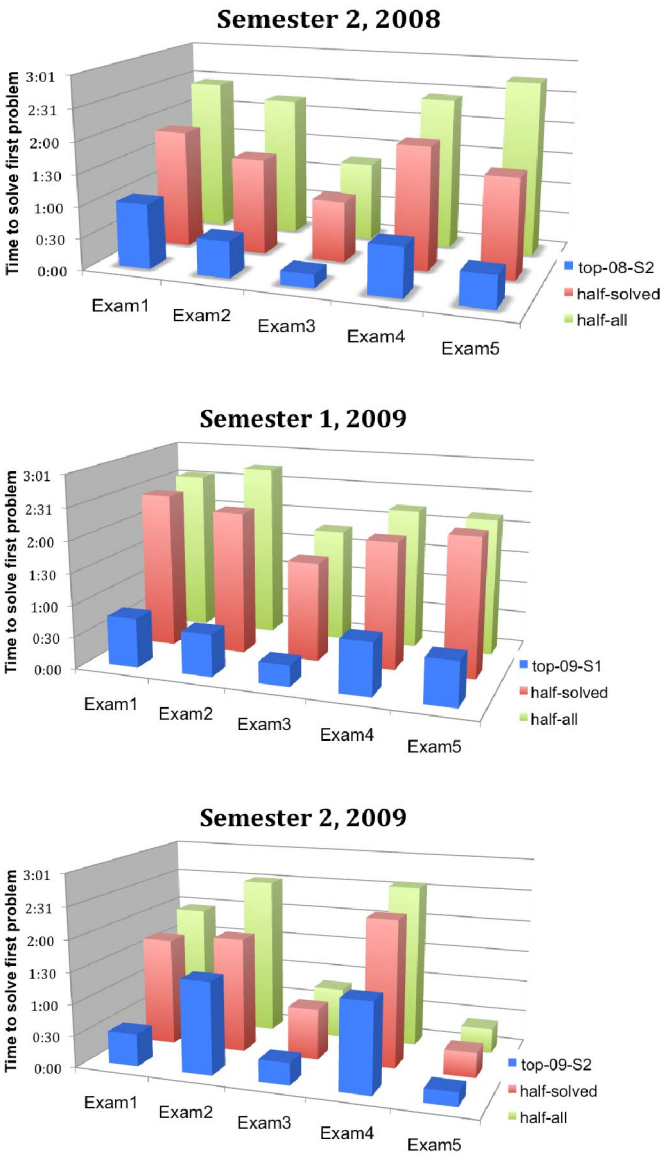

Figure 10. Time to solve the first problem at a practical exam, for 3 consecutive semesters.

In particular, they found hard dealing with complex input data that needed to be stored in another format to simplify coding of the solution. For example, the textbook and the course lectures cover graph algorithms such as Breadth Search First in detail. In the last practice session many students could work out that the given problem could be solved by (1) building the graph and (2) finding the shortest path from a given node to another. But only 3 out of 10 students were able to construct a graph from the given input information. Thus, we need to add more support material for the second part of the course to support coding for complex problems.

C. *Student feedback*

As expected, the better class performance in semester 2, 2009 is accompanied by positive feedback from the small group of students who took the course. We will report here on

some of the comment received and their suggestions for further improvement.

All students agreed the course have improved their problem-solving skills and their ability to work independently. Below are two quotes (printed with permission) from students that show we have achieved one of our main goals by providing the programming skills needed for their Master degree:

"During the initial stages I took 90-120 minutes to solve a single above problem. But now I take 30-60 minutes to solve the problems. SP eased me in doing assignments of Computer Vision, which are actually very tough to do"

"Although most of the tasks for this course have been finished, I still need to keep on practicing, since I have noticed that the practice we have done during the whole semester really helps me with coding and debugging"

All students thought the practical sessions have a good mix of problem types and they were positive about working is a group. One student suggested including peer-review on their coding as part of the group work. Antoher student made the following suggestion:

"My only problem was that some of my group members wouldn't attend group meetings. The course could have a forum that listed weekly group attends, and how many times you met outside your Practice Sessions to work on problems. That way, people might feel more accountable to attending group sessions".

During the course, most students reflected on their weekly reports on their need to work on pen and paper to check if their idea worked before coding. In the course feedback, a student reported on the importance of developing an algorithm first:

"The real power of the problem solving process comes down to pen and paper - first test out the examples on paper, by trying to "solve" the problem. Then try and figure out what you are doing to solve the problem. Make sure that you have examined all the EXTREME cases, base cases, boundary cases, before you implement it."

However, during exam time some students skipped this step and coded straight to the terminal with variable success. Thus, we need to do more work to reinforce this concept.

VI. Conclusions

We believe the strategies used in this course, based on aspects of deliberate practice [4], would be applicable to other technical courses with mixed cohorts.

Students are more likely to improve their skills if we provide clear expectations of what is needed to reach each grade level, and encourage the practice needed to achieve good grades.

It is important to assess students' initial skills in the first week, in order to help target support for the weaker students. We should note the review of basic techniques and debugging skills is also useful for stronger students, as it encourages self-reflection and good practice. Most students respond positively when they see some improvement early on the course.

By identifying the factors that cause failure and monitoring student performance, we could provide a tailored process for each student to achieve their objectives. The group reports provide good feedback on individual progress and chances to tailor their practice but this may be too onerous on the lecturer for a large cohort. As one of the main benefits of the reports was self-reflection, we will consider using blogs as a tool for students to report and reflect on their progress.

Future work on this course will focus on bringing more aspects of deliberate practice into the learning process including the selection of more questions designed to find and address particular skill deficiencies and more infrastructure to lecturers to monitor each student's progress and, just as importantly, to allow students to monitor their own progress.

Acknowldegments

The authors would like to thank their fellow lecturers Katrina Falkner, Peter Kelly, and Joseph Kuehn who helped us work through all the instances of this course. We also thank the supervisors who provided valuable feedback on the practical sessions.
Aspects of course development are part of an ongoing project to develop a problem-solving curriculum sponsored by a Google Research Award. Finally, we'd like to thank the people at TopCoder for letting us use their questions.

VII. References

[1] O. Astrachan, "Non-Competitive Programming Contest Problems as the Basis for Just-in-Time Teaching", 34th ASEE/IEEE Frontiers in Education Conference, 2004.

[2] Topcoder Inc., "Topcoder," `http://www.topcoder.com`.

[3] R.E. Slavin, "Research on Cooperative Learning, What We Know, What We Need to Know", Contemporary Educational Psychology, 21,43-69, 1996, Elsevier.

[4] K.A. Ericsson,"The Influence of Deliberate Practice on the Development of Superior Expert Performance", The Cambridge Handbook on Expertise and Expert Performance, 685-706, Cambridge University Press,2006

[5] Ericsson, KA (1993). "The role of deliberate practice in the acquisition of expert performance". Psychological review (0033-295X), 100, p. 363.

[6] Sonnentag, S (1998). "Expertise in professional software design: A process study". Journal of applied psychology (0021-9010), 83, p. 703.

[7] Specialised Programming additional course materials available at http://www.cs.adelaide.edu.au/~brad/educon2010/

[8] Slavin, Robert E. (1983). "When does cooperative learning increase student achievement?". Psychological bulletin (0033-2909), 94 (3), p. 429.

[9] James A. Shepperd, Social Loafing and Expectancy-Value Theory Personality and Social Psychology Bulletin, Vol. 25, No. 9, 1147-1158 (1999)

[10] McCabe, D. 2005. Cheating among college and university students: A North American perspective. International Journal for Educational Integrity 1 (1). Archived at http://www.webcitation.org/5Z5orfuVD

[11] David Barnes, Sally Fincher and Simon Thompson, Introductory Problem Solving in Computer Science, CTI 5th Annual Conference on the Teaching of Computing, Dublin, August 1997.