Learning About Software Development – Should Programming *Always* Come First?

Margaret Hamilton, Liz Haywood
School of Computer Science and Information Technology
RMIT University, City Campus
PO Box 2476V, Melbourne 3001, Victoria
[mh,liz]@cs.rmit.edu.au

Abstract

The issues surrounding curriculum design of many Computer Science and Software Engineering degree programs[1] are many and complex. In particular, the question of whether prior programming knowledge has any bearing on a student's success in learning and applying techniques for Software Analysis and Design is largely unresolved.

We undertook this study because as part of the continuous development of our degree programs, curriculum changes are always on the agenda. Anecdotal evidence suggested that students who had some programming experience were better equipped to perform Software Analysis and Design activities, especially low-level program design.

We surveyed a cohort of students to discover their prior programming experience and their perceptions of the advantage such experience gave them when undertaking a Software development project. We also considered the students' success in an introductory Software Engineering course (i.e. their final result) against their programming experience.

The results indicate that prior programming experience is not necessary for a student's success in a course that expects them to undertake analysis and design activities for a large-scale software product.

Keywords: Software Engineering, Programming, Curriculum Design

1 Introduction

Software Engineering Analysis and Design is a core course in all our Computer Science Programs. However the decision to teach large-scale program design to students who have no prior knowledge of programming (or algorithm design) is still a hotly debated question in our School. Currently we have two courses catering for the various cohorts of students from at least five different programs. One course is targeted at second year undergraduate students, who have completed a full year of study, which includes at least two programming courses covering various topics including generic algorithm design and coding using an Object-Oriented Programming (OOP) language, Java. In this course, the students are taught low-level design and implementation techniques before undertaking any analysis and design activities associated with large-scale software system development

The other course is aimed at Postgraduate students who are undertaking a Graduate Diploma or a Masters by Coursework program. These students do not necessarily have any programming experience and it is often the first course that they undertake within their program. Also, the first programming course they take involves the C programming language and so very few of these postgraduate students would have any knowledge of OOP. In this course, the students are expected to undertake various analysis and design activities for a large-scale software system, to design stage only. The course also caters for undergraduate Information Technology students who enter their program with advanced standing, from TAFE or other similar background, but do not have the pre-requisite programming experience needed to undertake the undergraduate Software Engineering course.

This paper compares the merits of both courses and discusses some of the issues surrounding the need for a pre-requisite OOP course. Section 2 contains some background and previous investigations. In Section 3 we present our student-centred approach with an electronic survey and analysis of the survey results, while in Section 4 we discuss our preliminary results and in Section 5 present our conclusions.

2 Background

In their final report to the IEEE and ACM, the Joint Task Force on Computing Curricula (2001) recognised that "the programming-first model is likely to remain dominant for the foreseeable future". This report was presented in December 2001, and though they acknowledged serious problems were inherent in that approach, they stated that "no adequate resolution has emerged" and continued innovation and experimentation with alternative models must be encouraged. "Like the problem of selecting an implementation language, recommending a strategy for the introductory year of a computer science curriculum all too often takes on the

[1] A degree *program* is made up of a number of *courses*.

character of a religious war that generates far more heat than light." Hence they chose not to recommend one approach but proposed three implementations of a programming-first model and three that adopt an alternative paradigm.

The three programming-first models are termed the traditional imperative paradigm, the objects-first design and a functional-first approach that introduces algorithmic concepts in a language with a simple functional syntax such as Scheme. The three alternative approaches are a breadth-first approach beginning with an overview of the discipline, an algorithms-first strategy focusing on algorithms over syntax and a hardware-first model "beginning with circuits and building up through increasingly sophisticated layers in the abstract machine hierarchy."

It is clear from their definitions and program content that our School has implemented the objects-first design for our undergraduate curriculum and the breadth-first approach for our postgraduate curriculum. This is in keeping with the cohorts of students and the lengths of time available for them to complete their programs. For the postgraduate students, no prior knowledge of computing in their former degree is required but it is assumed they want to progress into an IT-related job and so require breadth of knowledge in a relatively short time. Postgraduate students are expected to reach a level and understanding equivalent to the undergraduates in only twelve months of fulltime study while the undergraduate student has three years. In the final six months of fulltime study, the Masters students are expected to undertake courses equivalent to Honours level.

Duley and Maj (2002) make the argument that giving a student a dictionary does not help them put together sentences and construct a paragraph. So providing students with programming syntax does not help them code sensible programs. Alternatively, it has been suggested to us, by colleagues, that until a student understands an infinite loop and the problems associated with such basic syntax errors, they cannot analyse a problem sufficiently well to achieve a useful outcome.

Bagert (1998) recommends teaching the basic software engineering concepts early in the curriculum and delaying the large project development for a later course. "Introducing the student to many of the elementary concepts of software engineering … allows for a greater depth, breadth and comprehension of the material covered."

In discussing these issues, we asked ourselves the question of how much of this relates to how we ourselves learned to program, and how much to the preferred teaching methods and language constructs of the day. We are all clear that our students are required to know both programming and analysis and design by the end of their program - it is merely the order in which they are learnt. Sometimes, as in the case of the part-time student taking one course per semester, it is not possible to take the courses concurrently. If this is the case, which should come first?

Burton & Bruhn (2003) argue that mathematical principles should be taught in conjunction with a procedural programming language. "A too heavy emphasis on computational techniques in a beginning course can distract students from developing their own ability to conceptualize and form abstractions." They conclude that many students dread taking an object-oriented programming prerequisite because, without understanding these procedural and mathematical basics, programming is too difficult.

Our postgraduate students start with the C programming language and then can optionally take Java or C++. According to de Raadt, Watson & Toleman (2003), Java and C++ are the languages in highest demand for industry as judged by job advertisements, scoring 30% and 29% respectively of the 424 advertisements for programmers. C ranked fourth with 17% so our graduates are well placed for industry selection regardless, but perhaps they could do away with C and concentrate on the object oriented programming language.

Godfrey (1998) recommends devoting several lectures to object-oriented programming, also covering graphical programming and scripting languages and tools such as purify and quantify.

Cusick (2000) writes that one of the lessons learned from teaching software to adult students was that "early exposure to the fundamental concepts of design, testing, and planning are essential." However, it is perhaps the maturity of the students that enables them to understand the concepts and provide a more dynamic, self-directed learning experience. McCracken (2002) focussed a study on procedural experts learning OO design, and concluded, "...if we don't give students the opportunities to design relatively complex systems, they won't develop the metacognitive and domain independent designing skills necessary to solve those problems." So students learn to design by practice and not rote learning of prescriptive design principles.

In their study of software engineering best practice, Bernstein, Klappholz & Kelley (2002) recommend giving students a software project which is doomed to fail. They devised an instrument to measure both attitude toward and knowledge/understanding of software engineering best practice and showed that the "Live-Thru Case History" motivated the need for "rigorous requirements engineering, up-to-date documentation, identification of risks and development of contingency plans and evaluation of a design's complexity in order to manage cost, personnel and schedule".

We decided a useful starting point would be to survey our own students who had recently completed the course in an effort to better understand their perceptions.

3 The Survey

We sent an electronic questionnaire (approved by the RMIT University Human Research Ethics Committee) to our two cohorts of students who completed their course in semester 1 2003. Lethbridge (2000) noted that most

software engineering participants preferred the electronic survey, though they had also provided paper versions.

The group of students consisted primarily of postgraduate students in the first stage of their program, some were just starting out, whilst others had one semester of programming in C. However, there was a small cohort of undergraduate Information Technology students who had been given one year of exemptions in their undergraduate program, and may not have completed any object-oriented programming course.

3.1 The Questions

The first ten questions we asked, listed in Table 4-1, were straightforward and required the students to click a button against one of four or five possible responses. For example, for Question 1:

Before this course, how would you have rated your programming skills?

The possible responses were:

Very good, Good, Average, Poor, Very poor

and for Question 4:

Do you consider your past programming experience helped you in the course?

The possible responses were:

Very much, A fair amount, A little bit, Not at all

The last ten questions were more searching and probing. Several required longer student responses than simply clicking a radio button, for example:

"*Please comment on whether or not you think students should have a knowledge of programming before taking this course."*

3.2 The Responses

Many students welcomed the opportunity to talk about the course and completely filled the optional comment boxes with useful and supportive feedback. It turns out that most of the students who responded had done well in the course, only one student who had failed replied. We took this to mean that those who were successful felt more empowered to share their experience and perceptions. Forty students replied from the two hundred emails sent out. Of these, the majority were male (25), postgraduate (33), fulltime (34), international students (30), which is a good sample, typically representative of the class.

In order to determine the effect of previous knowledge of object oriented programming on their results, we first examined their results.

Figure 3.2-1 shows the students (1 – 40), sorted by result so that student 1 was the student who had not completed, student 2 had failed, the next 5 students had received PA, the next 13 - CR, 16 - DI while student 40 was one of the 4 who had scored a High Distinction.

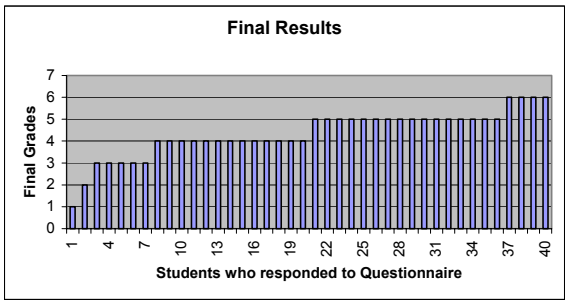

Figure 3.2-1 Student responses sorted by final result

We then plotted the students' responses to several questions against this grading of their results in order to investigate the effect on their results.

The responses to the question: *"Do you consider your past programming experience helped you in the course?"*, where the student had clicked a radio box scoring a 4 for "Very Much", a 3 for "A fair amount", a 2 for "A little bit", or a 1 for "Not at all", were plotted in the same order of ranking, i.e. student 1 is the same student as on the previous graph. These results are plotted in Figure 3.2-2.

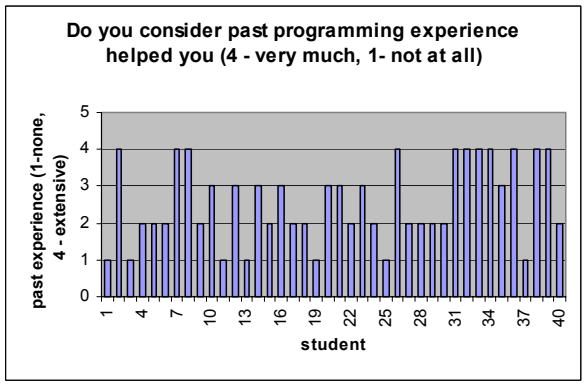

Figure 3.2-2 Student responses about past programming experience, in ranked order by result

As can be seen from the graph in Figure 3.2-2, there is very little relationship between a student's belief that past programming experience is useful and their final result. We noted that those who answered 1 or 2 make up the small majority which means that overall they think past programming helped only a little bit or not at all, while the minority believe it helped. This result surprised us immensely, as the academics who are arguing for the change of pre-requisites definitely believe any programming experience helps.

We also plotted what the students would recommend to future students - *"On a scale of 1-5 how necessary is a knowledge of object oriented programming for future students?"* Here the students could click a radio button corresponding to 5 for "Essential - it should be a pre-requisite", 4 - "Necessary - it is helpful", 3 - "Useful - it is helpful but students can get by without it", 2 for "Not useful - students can do well without it" or 1 for "Unnecessary - it confuses the situation and students would be better off without it". These results can be seen in Figure 3.2-3

Only one student each answered 1 or 2 while the majority answered 4, so it appears that these students would recommend other students to study an object oriented programming language first.

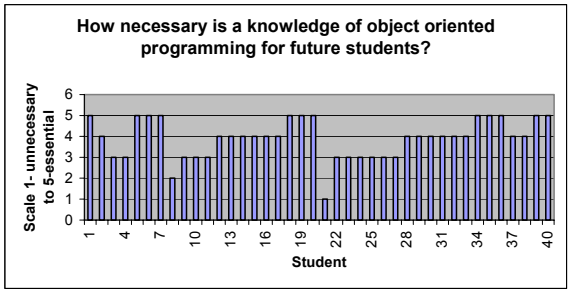

Figure 3.2-3 Student responses about recommending OOP for future students, ranked by result

3.3 The Effect of Previous Knowledge of Object-Oriented Programming

Another interesting question to investigate what these students consider they now know about analysis and design and programming. One question we had asked early in the questionnaire was what was their previous knowledge of object-oriented programming - *"Before this course, how was your knowledge of object oriented programming (eg Java or C++)?"* and the sliding scale from 1 to 5 respectively was very poor, poor, average, good, very good. Figure 3.3-1 depicts the perceived previous knowledge of OOP.

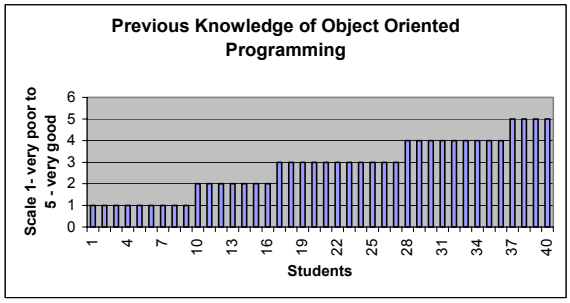

Figure 3.3-1 Students responses sorted on perceived previous knowledge of OOP

We plotted their final result based on their previous knowledge of Object-Oriented programming and found very little relationship, see Figure 3.3-2 below.

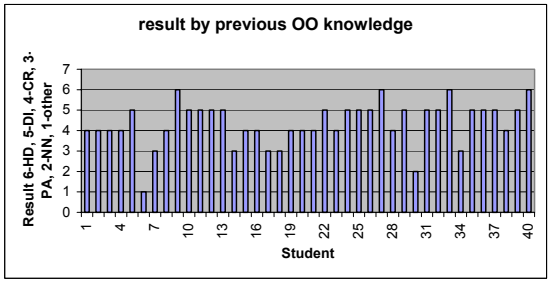

Figure 3.3-2 Students results ranked by perceived previous knowledge of OOP

Student 5 who had no previous OOP achieved a DI, while student 6, also with no OOP withdrew, and student 7 achieved a Credit. Student 30 with good OOP skills passed while students 33 and 40 achieved a HD.

However, when asked the question *"On a scale of 1-5 how do you rate your programming skills now?"* The following results in Figure 3.3-3 were obtained.

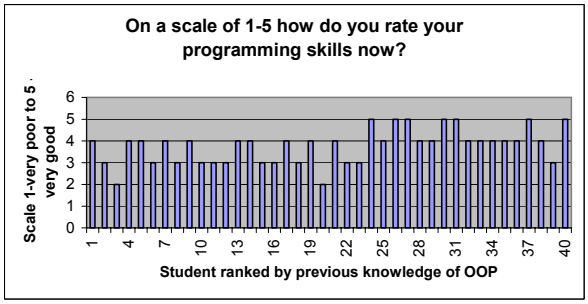

Figure 3.3-3 Students' rating of their *programming* skills on completion of course, ranked by perceived previous knowledge of OOP

Programming is always a very difficult skill to judge and when asked about their analysis skills, the students were slightly more confident: On a scale of 1-5 how do you rate your analysis skills? These results are plotted in ranked order of their perceived previous knowledge of OOP (Figure 3.3-4).

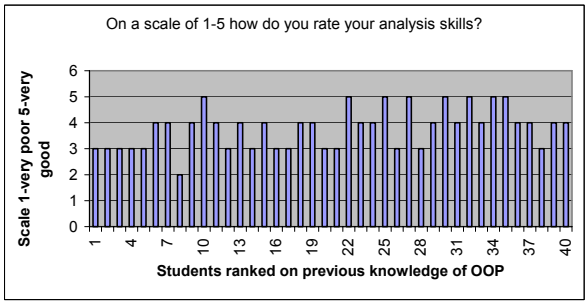

Figure 3.3-4 Students' rating of their *analysis* skills on completion of the course, ranked by perceived previous knowledge of OOP

They were definitely more confident about their design skills which were the ones the academics in our School were most concerned about (Figure 3.3-5).

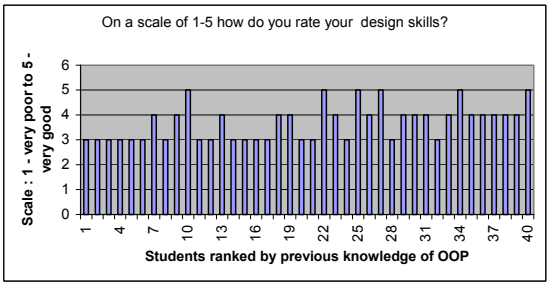

Figure 3.3-5 Students' rating of their *design* skills on completion of course, ranked by perceived previous knowledge of OOP

3.4 The Student Comments

Most of the students who responded to the questionnaire completed every question, although after the first 10 questions, it was suggested that the last ten questions were useful but not essential. Possibly it was because this was only a short questionnaire and most of these students had done well in the course, that they welcomed this opportunity and responded positively on the whole. Comments ranged from suggestions to change the textbook; *" text book. (real crap) "*; to shorten the exam and to change the assignment presentation. Most students had enjoyed the course and respected the lecturers, but there was one comment concerning a lab assistant who didn't know about Rational Rose which concluded: *"Maybe you should add a prerequisite for that: ALL the lecturers and tutors and assistants (who are high experts in OO world) must be able to explain the terms to an individual who DOESNT know any programming language"*. This suggests that we do have to explain basic OO programming principles to students who are studying OO analysis and design.

There were some very funny comments:

"This course is VERY time intensive, some acknowledgment of that I think would be good (by acknowledge I mean change ;)."

"*I think most students had done at least some procedural C programming.*"

"*We got our head around object, but that was only because one of our team was doing Java that semester. The other two (including me) were hampered by a good knowledge of C which allowed us to effectively invert the object design. Give some thought to the limitations of a primer in 'C' which is a likely background for many."*

4 Some Analysis of the Replies

Three separate measures were considered for each question: (a) The mean score and the standard deviation for each question; (b) the percentage of participants who gave each topic a score of 4 or 5 (meaning they considered their knowledge of this topic either good or very good); and (c) the percentage of participants who gave the topic a score of 2 or less (meaning their knowledge was poor or very poor).

The results are in Table 4-1, along with questions. We have labelled Column 4 Depth, referring to the depth of knowledge the students perceive, being a 4 or 5 for good or very good, and Column 5 for Weakness referring to the scale of poor or very poor which the students rated their knowledge for these particular questions.

It is interesting to note that in column 5, from question 5 onwards, the students didn't perceive weakness in their knowledge. However, for question 4, the majority of students had perceived weakness in their programming skills before the course.

No	Question	Mean	StDev	Depth	Weakness
1	Before undertaking this course, how would you have rated your programming skills?	3.3	1.2	21	9
2	Before this course, how was your knowledge of object oriented programming (eg Java or C++)?	2.8	1.3	13	16
3	Before this course, how was your knowledge of procedural languages (eg C, Pascal or COBOL)?	3.2	1.2	18	12
4	Do you consider your past programming experience helped you in the course?	2.6	1.1	11	21
5	Now, after completing the course, how would you rate your present understanding of objects?	3.8	0.9	23	2
6	How do you rate your knowledge of Unified Modelling Language (UML) ?	3.8	0.7	27	0
7	On a scale of 1-5 how do you rate your analysis skills?	3.8	0.8	25	1
8	On a scale of 1-5 how do you rate your design skills?	3.7	0.7	22	0
9	On a scale of 1-5 how do you rate your programming skills now?	3.8	0.8	26	2
10	On a scale of 1-5 how necessary is a knowledge of object oriented programming for future students?	3.9	1.0	27	2

Table 4-1 Statistical Analysis of the first 10 Questions

5 Conclusions

As noted in Column 5 of Table 4-1, after the course the students on the whole, do not perceive weakness in the areas of analysis, design and programming. This is in spite of their skill in programming which, when they entered the course, they admitted on the whole was weak (the score of 21 for Question 4 in Column 4 is more than half of the sample taken), and regardless of the fact that these students are in the early stages of their overall program.

Most of the students did write comments in the textboxes at the end of the survey, and their comments were evenly distributed between those recommending an object-oriented programming language be taught before this course and those not. Hence the students are as equally divided on this issue as the staff.

One aspect the students did agree on, was that the course involved a lot of work. There are many case studies and examples and content material to more than adequately cover the needs of this course.

Whilst we recognise that this is not a rigorous evaluation of the issue, it does alleviate some of the concern we have over teaching OO analysis and design of large-scale systems to students who have little or no OO programming experience. The students' responses suggest that it would be easier if they did study OO programming first, but that they were not unduly disadvantaged if they hadn't.

Whilst we respect the position of many of our colleagues, that students need to know how to program before they can possibly analyse a problem and design a software solution, we have shown that this is not the case – at least for success in this particular course. In fact, we would argue that in order for students to at least attempt design before coding, then the coding aspect of an assignment should be removed. We all know how tempting it is for a student to "do the code first" (and then draw up the design....). In our opinion, this does nothing to encourage students to look at the big picture – something very necessary for large-scale software development.

6 References

Bagert, D. (1998): A Model for the Software Engineering Component of a Computer Science Curriculum. *Information and Software Technology* **40**: 195-201.

Bernstein, L., Klappholz, D., Kelley, C. (2002): Eliminating Aversion to Software Process In Computer Science Students and Measuring the Results. *Proceedings of the 15th Conference on Software Engineering Education and Training (CSEET'02)*: 90-99.

Burton, P., Bruhn, R. (2003): Teaching programming in the OOP Era. *The SIGCSE Bulletin* **35**(2):111-114.

Cusick, J. (2000): Lessons Learned from Teaching Software Engineering to Adult Students. *Proceedings of the 13th Conference on Software Engineering Education and Training (CSEET'00)*: 39-46.

de Raadt, M., Watson, R., Toleman, M. (2003): Introductory Programming Languages at Australian Universities at the Beginning of the Twenty First Century. *Journal of Research and Practice in Information Technology* **35**(3):163-166.

Duley, R. and Maj, S. (2002): Cutting Hacking: Breaking from Tradition. *Proceedings of the 15th Conference on Software Engineering Education and Training (CSEET'02)*: 224-233.

Godfrey, M. (1998): Teaching Software Engineering to a Mixed Audience. *Information and Software Technology* **40**: 229-232.

Joint Task Force on Computing Curricula (2001): IEEE Computer Society, Association for Computing Machinery: Computing Curricula 2001 Computer Science - Final Report - December 15, 2001.

Lethbridge, T. (2000): Priorities for the Education and Training of Software Engineers. *The Journal of Systems and Software* **53**: 53-71.

McCracken, W. (2002): Models of Designing: Understanding Software Engineering Education from the Bottom Up. *Proceedings of the 15th Conference on Software Engineering Education and Training (CSEET'02)*: 55-63.