Principles for Designing Programming Exercises to Minimise Poor Learning Behaviours in Students

Angela Carbone, John Hurst
Faculty of Information Technology
Monash University, Australia
{Angela.Carbone, ajh}@
Infotech.monash.edu.au

Ian Mitchell, Dick Gunstone
Faculty of Education
Monash University, Australia
{Ian.Mitchell,Dick.Gunstone}@
education.monash.edu.au

Abstract

In most introductory programming courses tasks are given to students to complete as a crucial part of their study. The tasks are considered important because they require students to apply their knowledge to new situations. However, often the tasks have not been considered as a vehicle that can direct learning behaviours in students. This paper aims to encourage academics to start thinking about the tasks they set, in particular it explores characteristics of programming tasks that affect student learning and understanding in a first year undergraduate course as part of a degree in Computer Science at Monash University. Attention is paid to features of programming tasks that led to three poor learning behaviours: Superficial Attention, Impulsive Attention and Staying Stuck. The data gathered for this study which describe the students' engagement in the tasks are provided by students and tutors. The paper concludes with a list of generic improvements to be considered when formulating programming exercises to minimise poor learning behaviours in students.

1 Introduction

The paper stems from a study that commenced in 1995 at Monash University, aimed at tackling perceived problems in the teaching and learning of first year programming. The main concerns were high failure rates, a low flow of students into higher degrees and a perception of a wide variation of teaching skills.

The project was restricted to studying the teaching practices in two Departments of the Faculty of Information Technology (FIT), by a team known as Edproj. Edproj comprised staff from Information Technology and the Faculty of Education, and the team conducted research on the nature of student learning, teaching and teaching situations within FIT. From the study [6, 7] it was found that a number of aspects relating to task design were worth exploring, especially in relation to the nature of good learning. As a consequence, a further study continued to investigate the set tasks given to students studying first year programming as part of a degree in Computer Science at Monash University. The intention was to determine whether there were features in tasks that led to poor learning tendencies in students and if so, develop a set of guiding principles that might assist teachers in designing programming tasks effective in improving students' learning and understanding.

The purpose of this paper is to encourage academics to think in terms of student learning when they are designing tasks, and to alert them to the dangers of the poor learning tendencies that students may adopt when inappropriate control over the learning process occurs.

2 The Edproj Investigation

The Edproj project gathered data [3] that showed that the majority of students' learning takes place in practical classes and in tutorials. Practical and tutorial classes are the places where students attempt and complete many problem solving exercises. The experience at the time showed a common paramount emphasis by the students to get the set tasks done, meet the requirements as they saw these and then stop. Tutors reported that it is "tremendously difficult to get students to do something beyond what was set". Concepts which were not to be assessed were ignored. Tasks intended by lecturers to be gateways to exploration and reflection were seen by students merely as hurdles to be negotiated. The students saw the course to be about getting programs to compile and did not see the abstractions intended by the lecturer; in fact often these were obscured behind the high task demands of coding.

The Edproj investigation at Monash indicated that students did not reflect on what they did and did not understand, and lacked strategies for improving their understanding. Yet most lecturers, demonstrators and tutors indicated that they would greatly prefer students to take a thoughtful approach to their work.

3 The study

The initial Edproj investigation indicated the value, to academics, of studying student learning in a programming discipline. The comments made by lecturers, tutors and students, were the driving force behind the exploration into aspects of programming tasks that affect student learning. This section begins by describing typical types of tasks set as part of the course studied, followed by how data were collected to determine the effect of the tasks on student learning.

3.1 The programming unit and its tasks

In first year Computer Science, at Monash University the academic year is divided into 2 semesters. The subject taught in semester 1 covers introductory programming concepts and a general introduction to computers. The subject in semester 2, called Computer Science, is divided into two parts: Part A focuses on Algorithms and Data Structures and Part B focuses on Computer Systems.

The established practice in Computer Science was that the lecturers presenting a subject would develop a set of tasks for tutorials and laboratories. In the practical classes, students are required to complete sets of programming tasks deemed appropriate at that stage of study. Each task is assessed. The students work in a lab with the presence of a demonstrator employed to assist them with their programming difficulties. In the lab, students have to produce the pre-lab work for assessment and submit the other tasks in the lab for assessment. The tutorials were conducted in rooms without computers, and used to generate student discussion as students worked through the exercises. In tutorials, students were expected to offer ideas, make speculations, give alternative explanations or offer answers to pre-set questions. No assessment of their work was made in the tutorials.

Appendix 1 provides examples of typical programming tasks labelled T1 to T8, set under each component of the course. These tasks will be referred to in section 5 of this paper.

3.2 Data Collection

Three approaches were used to determine whether the tasks affected student learning. In 1997, data was gathered from students who were interviewed on a weekly basis as they worked through their tasks. In 1997 and 1998 tutors reported on how students progressed on the tasks as they assisted the students working in the labs and tutorials [2], and in 1998 cases were written by students describing their learning experiences and engagements in the tasks.

3.2.1 Student Interviews

In semester 2 of 1997, eight students from 315 were selected at random to participate in weekly 20-30 minute interviews. The interviews were conducted during the lab session, usually an hour or two into the lab. The students were guaranteed anonymity, and extracts from the interviews are presented in section 5.

There was one problem with this approach. Often students lacked the vocabulary to describe their engagement in a task, which made it difficult to explore their learning on the current programming tasks. As a consequence, a short course described in section 3.2.3, was designed to provide students with frames to analyse and describe their own learning.

3.2.2 Tutor Perceptions

A web based database application called TeachTools [2] was partly designed and implemented for tutors to provided reviews of the set tasks for later analysis. Tutors were required to respond to a list of questions which were aimed at highlighting which parts of the task were helpful in aiding students' learning and understanding and what concepts were made unclear to the students. The tutors provided feedback on the tasks given in the Data Structures and Algorithms component of the course, constantly modelling how thinking about the task and how issues involved in learning could make the task far more informed and purposeful. Tutors also offered suggestions on how to improve the task sheet.

3.2.3 Student Cases

A third source of data came from eight students who voluntarily participated in a short course on "Helping Learners Learn". The course was conducted in the first four weeks in semester 2 of 1998 and consisted of two one-hour sessions per week, students explored aspects of the tasks and their own learning that might have affected their progress.

The course covered deep and surface approaches to learning and factors that promote their use [4,5,8]. Poor learning tendencies were discussed, and students were given the opportunity to reflect on their own use of cognitive strategies in learning contexts. The course also provided students with some knowledge about learning theories, ranging from behaviourism, through cognitivism to constructivism.

It was intended that this mini course would be attractive to students who wanted to improve their learning as well as provide them with frames to analyse and describe their own learning. Surprisingly, students that volunteered had received high marks (in the range of 78%-91%) for their semester 1 exam. It was therefore, presumed that these students would have very good learning behaviours.

4 Poor Learning Tendencies

The fundamental underpinning that framed the data collected and issues extracted in this investigation was a study into student processing habits by Baird and Northfield [1]. Baird identified of a series of processing habits in tertiary learners that he called Poor Learning Tendencies (PLTs). He pointed out that PLTs were all characterised by a passive, dependent, uniform approach to learning. The learner may well be keen to succeed, but sees his or her role as a passive recipient of information and

instructions; the teacher (or text) will tell the student what to do, how to do it, when to stop, what questions are worth investigating, what examples are worth considering and even whether or not they understand the work. The processing habits identified by Baird seemed an appropriate starting base to determine whether the tasks led students to develop poor learning tendencies. Of the processing habits found by Baird that act as barriers to achieving quality learning, three are considered in this study. They include:

1) Superficial attention. This involves skimming over a communication, with no attempt to actively process the information in order to generate personal meaning.

2) Impulsive attention. Some parts of a communication are attended to but others are overlooked. For example, the learner may focus on a interesting example and ignore a major point.

3) Staying stuck. Lack of any strategy to cope with getting stuck except to call for help. No attempt to return to the instructions, reflect on the strategy selected, analyse what has been done so far or consider alternative approaches.

This group was chosen because the PLTs cluster reasonably well together in relation to students' monitoring of their learning. Other PLTs will be discussed in a later paper.

5 Results

The three sources of data taken from student interviews, tutor perceptions and student cases will be referred to as SI, TC and SC respectively. Subscripts are used to identify the different responses and analyse their content. This data is used to extract common features of tasks that led students to adopt one of the following poor learning tendencies; Superficial attention, Impulsive attention, and Staying stuck.

5.1 Superficial attention

5.1.1 Evidence of Superficial Attention

Task T6, listed in Appendix 1, is given to students as part of the Data Structures and Algorithms component of the course. The intention of the task was to give students the opportunity to familiarise themselves with stacks and experience in implementing these. The tasks required students to be able to reproduce what had been specifically covered in the lecture with minor modifications.

Interviews with students revealed that students didn't search for a deep understanding, they were happy to apply their attention superficially and succeed by copying slabs of code. Even though the task asked students to modify code, students copied the code without thought. Tutors noted that many of their students missed the main concepts and their programs were replicas of those provided by the lecturer. Below are typical types of comments made by students and tutors for tasks T6, T8 and others.

SI_1	*I copied the program to create a stack from the lecture notes. I couldn't understand "pop()" and I didn't understand the difference between "s->top and s.top"*
SI_2	*I had managed to get the first four questions working simply by copying sections of the code from my lecture notes. I didn't really understand what was happening to the frame pointer, stack pointer, parameters and return values. I could just see that it worked, somehow.*
TC_1	*Concepts were lost because students weren't expected to write any of the sorting procedures themselves, most copied the code straight from the lecture notes.*
TC_2	*Students copied straight out of the lecturer notes without getting the stack concept*

5.1.2 Task features leading to Superficial Attention

There was one strong feature inherent in some of tasks that led to superficial attention. Tasks that require reproducing or modifying code often do not gear the students into discussions to think deeply about the material, or the type of learning one may have hoped for. Often there are reasons that explain why students don't actively process the information to generate personal meaning. The comments above show that students managed to complete the work and succeed by simply copying directly from the lecture notes. In the case extract below, a student describes how at the commencement of the task s/he is keen to succeed, and is using self-questioning to aid in the understanding of the material. S/he spends some time trying to understand the problem at hand but then admits that the task was taking too long to complete and the pressures of assessment were setting in. These were seen as more important than searching for meaning and understanding of the key ideas.

SC_1 *Gingerly I opened the notes to the page holding the relevant example. At a first glance, I was horrified at the complexity of it all. Why were they using that register? Why that location on the stack? What did that notation mean? My earlier warning calmed me slightly: "first impressions..." Taking a deep breath, I went through each step of the sample code, extracting all the information I could.*

After a good fifteen minutes, the underlying sense of it all was not sitting quite nicely, as I would have hoped. The sample code seemed to be making sense, but I was struggling to gain an overall picture. I considered spending more time looking at my notes. But time was running short and I had a prac to finish.

In the hope that my actually implementing the code would concrete the concept, I dove straight into it... but hit the ground very quickly. Now I was getting impatient. "That's it", I thought, "I will do this the crude way. I will copy the notes." So that is precisely what I did.

5.1.3 Improving the tasks

Ultimately lecturers teaching programming would expect their students to be able to design, implement and test a relatively complex piece of software. It is a common belief that the larger and more complex the code students write, the better programmers they will be. Most programming

courses are designed to include a laboratory component in which students spend 2-3 hours coding the tasks set. However, if students are asked to write complex pieces of code at too early a stage they can be pushed into adopting a poor learning tendency. It would be a complete change in culture for some academics teaching programming to design some tasks that do not require pure coding. Often smaller tasks can still be used to develop the students' interest and understanding of the subject. Below is a list of suggested improvements that can be made to tasks:

- **Not always coding.** Often students are required to write lines of code, and very rarely are students required to do alternative activities. Getting students to diagrammatically present material often highlights misconceptions that can be addressed immediately. Including tasks that require tracing code, or answering a series of questions, can be used as alternatives to purely writing code.
- **Rewards for understanding not completing.** If students were aware that understanding was rewarded, and not copious amount of code, they might be less likely to take a crude approach to completing the task.
- **Outline a method of attack.** Without a design students can wander from the intended pathway and ultimately reach a point where the only source of help is seen to be copying extracts of code provided.
- **Smaller coding questions.** Introduce questions that don't consume too much of the students' time, so they don't feel pressured into copying straight from the notes.

5.2 Impulsive attention

5.2.1 Evidence of Impulsive Attention

Task T1, in Appendix 1, contains an example of a programming task given to students as part of an introductory course on learning to program.

The intention of the task is to help students learn about functions. The task required students to design a solution to a mathematical problem using functions. In this case coding the solution was not as simple as reproducing the code given in lectures.

An important observation made by many tutors was that they noticed that their students were focusing on the wrong aspect of the task. Focus was placed on understanding how to preform the mathematics behind the calculation rather than concentrating on invoking a function, parameter passing and returning values. As tutors commented:

TC_3 Finding the powers of numbers is not a common thing student do very often. So they concentrate on getting the calculation right rather than how to use functions.

TC_4 As for q3, they definitely concentrated on the mathematics rather than programming. Therefore they are not learning what they should be, just spending a lot of time on the mathematics.

Tutors described how they watched their students spending the majority of their time on one question, leaving no time to attempt the other exercises, and how too much attention was placed on parts of the task that weren't important.

TC_5 Many of my students wrote an encrptWord function, and spent most of their time on that function, rather than the letter one. It might not be mentioned in the question, but should at least be made clear to demonstrator that the student needs to use gets() or fgets() for loading strings. Also the example should have upper case letters in it to show what would happen.

TC_6 Some students had problems with linked lists and many spent a great deal of time making menu generated user interfaces, which looked good but also wasted a lot of time.

Other tasks like T2, T3, T5, T7 (Appendix 1) lead to similar tendencies. Students describe how they spent more time on aspects of the task that weren't the key concept.

SI_3 I didn't like this prac, I did more thinking about cycling arrays, than actual pointers.

SI_4 I liked this question but many other students found it difficult to get the solution. The focus seemed to be more about loops than memory allocation.

SI_5 There was too much detail to comprehend everything at one time.

SC_2 I felt totally swamped by the mass of information flooding towards me. I knew on a very basic level what these things were, I even knew how to use some of them properly, but given that it was near the end of semester and exams were coming up, I simply did not have the time nor the patience to sit and work through everything to the depth which I thought was required to complete the project.

Often there are explanations as to why students miss the important points, and sometimes it isn't entirely their fault. In the two cases below students describe why they missed out on the key aspects of the task and the "real learning".

SC_3 Only 15 minutes left and I was almost in tears. I think the demonstrator felt sorry for me. Why wouldn't this damn program work! At this very instant I hated computers and their machine language.

This prac wasn't difficult; it was just long. Translating C code to MIPS means more lines of instructions to perform the same task, hence more typos, more bugs and more stress. The end of the prac was near, I was still on question 1, and was seriously about to cry. I had no idea what was wrong with my program, but to skip to question 2 would be

to open another can of worms and would only make me feel even more incompetent.

SC_4 *When I got into the prac and tried to run the program, I got an error message like "Error in 0x7808" which, as far as I could see wasn't an address that was being used by either my program or the SPIM emulator. I asked the tutor for help, but we were all needing help, so his time was limited, and as MIPS is such a low level language, problems don't just jump out like they do in C. He suggested that I step through the program, in which the program executes one line at a time, as the user presses 'Enter'. It was a drag, but after 45 mins of messing around, it didn't look like I had much choice. So I started. Then, because of the embedded loops, I realised that I had to do at least 450 passes of about 20 or 30 lines of code. I groaned, I banged my head against the wall. I was sure the code was fine, but I couldn't get any help and was forced to go through the loops, all the while watching the output screen, and about 6 or 7 registers at the same time. I eventually found the mistake - about 15 minutes before the end of the prac. The thing that really got me was that it was only the first part of the prac and the second part was so easy that I had it done it in 10 minutes, and so I 'passed' the prac - but I didn't even get to look at part 3, which looked to be where the real learning was at.*

5.2.2 Task Features leading to Impulsive Attention

Two key characteristics were found in tasks that lead student to impulsive attention: tasks that do not emphasise the key concept and tasks that contained too many unfamiliar concepts. Other tasks that were prone to long coding solutions and difficult to debug, although not leading to impulsive attention in the students, appeared as impulsive attention to tutors. The tutors observed students focusing on one question when in fact the student's time was consumed on debugging so they couldn't complete the remaining questions in class.

Often students are given a set of preparation questions that can be made useful to emphasise the key components of the task. This also helps the student select what material is important and needs attention. In the second prac of Data Structures and Algorithms students were expected to implement a reverse polish calculator. This required the student to understand reverse polish notation, file i/o, a number of C utility functions and how to design an algorithm to parse the data. The prac was aimed at helping students familiarise themselves with stacks and how to implement them, yet most students struggled with the first phase of the prac, which was designing the algorithm to parse the data. The magnitude of the problem caused the lecturer to make a substantial proportion of the code available to the students. Unfortunately there was an unintentional error in the code issued to the students. This caused some students to spend many hours searching for errors in the code they had written, when in fact the error was in the code provided. This case is described below.

SC_5 *We had to code this program that would read in a line of characters and return each with a description, either number, operator (+, -, /, *, or =), or invalid entry (anything else) and then perform the required function. We were to use a stack to do this, read them in and spit them out again. Part of the prac required the creation and manipulation of the stacks - popping, pushing, initialising etc. I did this blindfolded. I understood these things. But part of the prep for the prac had been to get this program to do the symbol recognition part - name the characters that were scanned in. We were given most of the code already, with 3 sections that read *Insert Missing Code*. So I get what they give us in, and decide to trial run it.*

Crash. I leave it, its late, it'll work tomorrow. Next day: Crash. I scan the code. There are a few functions I don't know, they look just like scanf, and a couple of '%g' that I don't get. So, against all better judgement, I go to the help menu. Instead of being useless crap like in windows, it's all gobbledegook written by techs. My understanding of scanf becomes complete confusion. So I let it lie, and go back to what I know. I try to debug it with some print statements, to see where the program is going, what's happening where, etc. Nothing. I decide that it's time to get the text and go to the library on my way home. At home I fool around with the code in notepad, hoping to wing it because I don't have the compiler. I've done it before, and it's good practice for making sure you get the code right first time around, because otherwise all my homework time is for shit. The textbook is marginally more comprehensible, but little help.

I read heaps, and get little coding done, but feel more confident. It's starting to test my confidence in my ability and understanding of what we are doing.

It can be done, obviously. So I go back to it on the weekend. I make a special effort to get to a machine with a compiler at uni. I load it all up, feeling good, I checked it back to front at home. First time I run it, it doesn't only not work, it crashes my pc. I fool around with some of the code. It's compiling ok, but crashes the pc again. Six times in all, then I leave, pissed off, confused, confidence completely undermined, and forget about it.

One hour before the prac, in the class, the lecturer points out that the code provided has a mistake in it. Simple, but deadly. When I go to the prac, I fix this straight away, and hey presto, working code. But now I haven't actually finished the prep because of the problem with what we were given. I haven't even thought about the other questions, I've only just done the prep and question 1.

5.2.3 Improving the tasks

When a task appears as a set of independent packages, students often attend to some parts of the task while other aspects are overlooked. For example, with the millenium bug around the next corner, the date task, T2, has been a popular one. The date exercise is fairly difficult one for students as it contains many different parts with relatively complicated algorithms. If the focus of the task is about functions, less emphasis should be placed on the algorithm.

TC_7 *I've never been too happy with the date question. We need to split it into two parts. The first part should be an easy question with functions calling others. The second part should be similar but with a more complicated algorithm. In any case stress the functions, rather than the complicated algorithm.*

Often tutors modify the task to highlight the key concepts, for example, if the key concept is array access it may be easier to work with an array of integers than an array of strings in C.

TC_8 *I modified the exercise to work with an array of integers instead of strings. I thought that using strings would add an unnecessary complexity to the problem, and students would overlook the underlying concept.*

Below are two suggestions to minimise impulsive attention:

- **Emphasise the key point.** Usually there are many ways to code a solution to a problem. If the important points are emphasised in the tasks, through the task's aim and the type of question, impulsive attention might be minimised.
- **Provide adequate resources for the introduction of unfamiliar material.** Sometimes subjects are so tightly structured that it's not possible to cover everything in lectures. As a consequence, many lecturers introduce new material in the laboratory and tutorial exercises. Often it is not possible to remove material from the subject, leaving some of the material introduced for the first time outside of that environment. While the idea of introducing new material is fine, the resources needed to help students understanding is usually inadequate, rather than carefully planned.

5.3 Staying stuck

5.3.1 Evidence of Staying Stuck

Some tutors expressed disappointment in their students, they felt that the students were progressively becoming too dependent on them when they became stuck, for example:

TC_9 *Students were unwilling to attempt to solve problems without constant demonstrator guidance.*

TC_{10} *Unfortunately, I am extremely disappointed with all but a select few of my students. The underlying attitude seems to be "Why is this so hard?. I thought the answers would magically appear in my head."*

Cases of students describing where in the task they became stuck and what strategies they followed are listed below. The first case (SC_6) talks about a task which is given as a group assignment, the second and third cases (SC_7), (SC_8) discusses part of a MIPS programming problem.

SC_6 *The project is to write a program to play the computer game "Hunt the Wumpus". You are in a cave of a Wumpus. The Wumpus likes you very much especially for breakfast. To avoid being eaten, you must locate the Wumpus and shoot it with your gun. Further details of the cave layout, hazards and gaming strategies are given in the assignment.*

SC_6 *I was stuck. Pure and simply. I dared not ask the lecturer for help and no one I knew was working on the same part of the program as I was - even if they were, the general impression I received was that they were hiding their secrets with ferocity.*

I brought up the subject with other students. I was expecting a rush of complaints on how hard it all was - especially the shoot module, but I got nothing. 'Why is no one saying anything?' I thought, 'Is no one else finding this hard? Did we get the same project?' The other student's lack of concern forced me to plod on.

I came to the conclusion that I needed to seek help. There was no way I could think of solving the problem. It was a definite dead end. The only person who I could think of was an older friend who had quite a bit of experience with C programming. I described my problem to him and gave him the project sheet to look at.

The case above reveals the hesitation by the student to admit that they did not know how to proceed. This appeared to be a common problem amongst many of the weaker students. In this case, the student describes a group assignment given towards the end of semester 1. The intention of the project was to help the students learn to develop software in a team, and to integrate the various aspects of C programming learnt earlier in the semester.

The following two cases outline the students' engagement in a MIPS programming task, T8. The objective of this task was to learn how MIPS implements loops and arrays. The cases reveal how students reflected on what they had done and how they tried to help themselves, but when they had spent too much time making very little progress, if any at all, they turned to the tutor as their only resource for help.

SC_7 *Debugging was a nightmare. There were masses of code splashed all over the screen – all of which had began to look the same. Loops which I thought had made sense initially no longer did and I realised I was in trouble.*

Help was required. I asked my friend to check the code, see if they could find any obvious errors, but they just scoffed when they saw the mess in front of them.

"Why didn't you break it up into sections?" they asked. "You start with a small piece, test it, see if it works and then move on. There could be a million things wrong here".

My eyes began to burn at the thought of restarting. Instead, I attempted yet again to debug. Hours later, which turned into days later the problem had not shown itself. All I knew was there was a problem with one of the loops, but which one and where?

The day of the prac, I was somewhat relieved to find others had similar problems to myself, but that did not change the fact that my mission had gone uncompleted. The demonstrator was sympathetic and gave me a mark of 17 out of 20. I can remember thinking that I would not let this one prac question ruin the rest of the semester for me. I was waging a war against MIPS. This was only the beginning – I would not accept this defeat.

SC_8 *I loaded the whole program to firstly weed out the obvious syntax errors. I then, optimistically, tried to run it. Who doesn't check to see if its going to work first go?*

"Bad address at ...", "Bad address at ..."

just kept scrolling down the screen. I couldn't see what was causing this and neither could the demonstrator.

I spent ages by myself stepping through trying to find this bad address, but I couldn't see what was wrong. Finally the demonstrator noticed that the simulator wasn't letting me use register $s1 - I felt some sort vindication. I changed the registers. Still not quite working, but at least one problem was gone. I was starting to think I should have typed my code before the prac - it would have given me some more time for problems like this...

5.3.2 Task features leading to Staying Stuck

Data suggest that there are three points where students can stay stuck in tackling a programming problem. Either the student doesn't know how to begin tackling a task (case SC_6), or they don't know how to design a solution to a task in manageable components (case SC_7), or they become stuck at the debugging level after having attempted to code a full solution to their problem (cases SC_8).

Case SC_6 illustrates how a student felt when attempting to tackle a large programming exercise (T4). Students are scared of being labelled stupid if they can't follow part of the task, and many times they are deterred from attempting some questions because they know they can't solve them correctly and any incorrect attempt would cost them marks.

Case SC_7 illustrates how students lack strategies in designing solutions for fairly large problems. Often the only resources they are aware of are their text book, the lecture notes and their tutor. The lecture notes and text book usually contain complete programs without strategies to design, implement and test code in manageable chunks. By the time the student approaches the tutor for help it is usually too late in the solution development, and tutors, reluctant to follow large amounts of code that haven't been tested, offer suggestions that require students to tackle the problem using a completely different approach.

Cases SC_8 demonstrates a common event of students writing their solution (to tasks T8, and the like) without any testing along the way. As a consequence, students are prone to making errors and lack basic debugging strategies to help them complete the task.

5.3.3 Improving the task

Below is a list of suggested improvements that can be trialed to tasks to minimise staying stuck.

- **Tactics on how to start with graded helps.** Challenge a student first, don't explain everything.
- **Provide useful references and resources.** Often the only resource students are aware of are their textbook, lecture notes and the tutor.
- **Provide guidelines to writing and testing code in manageable chunks.** Include debugging strategies. Guides to writing code should be provided rather than providing the code. Build a program in stages, for example, if part of the problem requires file input and output, students should write a small program to ensure they understand how to do that part.

6 Conclusion

There are a number of ways in which programming tasks can be set. Some may be written and designed with the best of intentions yet can cause above average students to adopt some poor learning approaches in their study. This paper highlights three poor learning tendencies that tutors saw their students display in their classes and that students described in their attempts to complete programming exercises as part of their first year Computing Degree. The data suggest that the task set can gear students to different types of learning tendencies. Knowing this, it is important for academics to start thinking in terms of how to frame tasks so that good learning patterns are adopted and pursued by students and rewarded by assessment. For students that might not have a good programming framework to begin with, this is particularly important.

Throughout this paper suggestions have been made to improve programming tasks to minimise the possibility of students adopting one of three poor learning tendencies. At the very least, I hope this study alerts academics to the dangers of poor learning tendencies that students may adopt when inappropriate control over the learning process

occurs. It is intended that this paper encourages academics to think in terms of student learning when they are designing tasks. The next phase of the study will explore tasks that encourage good learning behaviours motivating students to adopt a deep approach to learning.

7 Acknowledgements

The authors wish to thank all the first year tutors and demonstrators for reflecting on their teaching, and reviewing the set tasks. Thanks are extended to the first year Computer Science lecturers for allowing their tasks to be reviewed. Special thanks are given to the students who participated in the weekly interviews and those who provided written cases of their engagements in the tasks.

References

[1]. Baird, R.J. and Northfield R.J., *Learning from the Peel Experience*. Melbourne, Australia: The Monash University Printing Services, 1995.

[2]. Carbone, A., Drago M., and Mitchell I., *Web Based Tools to Maintain Teaching Strategies and Resources. What works and why? The Fourteenth Annual Conference Proceedings of the Australian Society for Computers in Learning in Tertiary Education,* Academic Computing Services. Curtin University, Perth, Australia, 1997. pp 101-110

[3]. Carbone, A., Mitchell, I., and Macdonald I., *Improving teaching and learning in first year Computer Science tutorials. Making new Connections. Proceedings of the Thirteenth Annual Conference of the Australian Society for Computers in Learning in Tertiary Education*. Faculty of Health and Biomedical Science, University of Adelaide, Adelaide, Australia, 1996. pp 571-572

[4]. Gagne, R., *The conditions of Learning*. New York: Holt, Rinehart & Wilson, 1985.

[5]. Metacalfe, J. and Shimamura J.P., *Metacognition: Knowing About Knowing*. Cambridge, Mass: MIT Press, 1994.

[6]. Mitchell, I. and Macdonald, I., *Learning and Teaching in First Year programming FCIT/Education Faculty Research Project Report*, Monash University: Melbourne, Australia. p. 27., 1995 (available from author)

[7]. Mitchell, I., *et al.*, *Progress Report on the Education Project Group*, Monash University: Melbourne, Australia. p. 25., 1996 (available from author)

[8]. Shoenfeld, A., *Radical constructivism and the pragmatics of instruction.* Journal for Research in Mathematics Education, 23: p. 290-295, 1992.

Appendix

T1 - Tutorial Exercise Sheet 7 (Functions)

Write a C function which takes an integer parameter (greater than 1) and returns the highest power of that integer which is also less than 1,000. For example

Given the parameter 2 your function returns 512 (2^8)
Given the parameter 3 your function returns 729 (3^6)

T2 - Tutorial Exercise Sheet 7 (Functions)

Write a module for determining the day of the week on which your birthday will fall in any year, given the day of the week and the date on which you were born.

T3 – Practical Exercise Sheet 8 (Functions)

Design an algorithm that reads in a string S and number k (the key) and prints a new string (the cyhertext) obtained by encrypting the string S as follows:

Each alphabetic character is replaced by one which is k letters further on in the alphabet. (Eg if k=3 then a is replaced by d, b is replaced by e,..., z is replaced by c. If the input string is Computer Science and k=3 your program would print out the string frpsxwh...)

T4 – Practical Exercise Sheet 11 - 12 (Software Project)

This practical session aims to help you learn to write software in a team, and to integrate the various aspects of C programming that you have learnt. The project is to write a program to play the computer game "Hunt the Wumpus". Details are given.

T5 – Practical Exercise Sheet 1 (Structs and pointers)

Write a C program to read and write N CD titles and prices. Rewrite your program so that when it reads in a title and a price, it prints the title and price out and asks you whether you wish to change either. If you type "y" the program should then prompt you for the new title and price, and then make those changes.

T6 – Practical Exercise Sheet 2 (Stacks)

*Copy and fill in the missing code in the program on the next page. The program reads in characters strings, which have a maximum length of MAXSTRING characters and do not contain blanks, newlines or tabs. It then checks whether the string is a number or one of the following operators +, *, -, /, or =. If the string is a number it prints out NUMBER and the number which is read in, etc...*

Modify the code given in lectures, and in your text book, to implement a stack which can hold numbers of type double.

T7 – Practical Exercise Sheet 3 (Dynamic Memory)

Consider an array of x integers. In various pattern matching algorithms procedures one needs to find the subarray whose sum is the maximum. If the array consists of only positive integers, then this subarray would be the whole array. The problem occurs when some of the integers are negative. Write a program which for a given array of integers, finds the maximum sum of its subarray.

T8 – Practical Exercise Sheet 4 (Loops, Arrays)

Given the following bubblesort C program (code is given) *write a MIPS program that faithfully implements this program. Comment your code.*