It's Not Them, It's Us! Why Computer Science Fails to Impress Many First Years

Rashina Hoda[1] **Peter Andreae**[2]

[1]Electrical and Computer Engineering, The University of Auckland, New Zealand

[2]School of Engineering and Computer Science, Victoria University of Wellington, New Zealand

r.hoda@auckland.ac.nz peter.andreae@ecs.vuw.ac.nz

Abstract

High attrition and failure in first year computer science and software engineering courses has often been linked to the personal traits and skills of students – dividing the world into those that "get it" and those "that don't". We present several concrete strategies based on the recently developed Learning Edge Momentum (LEM) theory, which when applied together, were found useful in reducing failure rates. Based on the our experiences, we challenge our current understanding of attrition and failure in first year courses and dare to claim that maybe it's not them, it's us that is the problem.

Keywords: computer science, software engineering, first year course, attrition and failure rates, LEM theory

1 Introduction

Attrition and failure in first year computer science and software engineering courses has often been linked to the personal traits and skills of students, sometimes referred to as the "geek-gene". According to this notion, the world can be divided into those that "get it" and those "that don't". In light of recent research emerging from the University of Otago, New Zealand (Robins, 2010), we attempt to redefine our current understanding of attrition and failure rates in first year courses.

The Learning Edge Momentum (LEM) theory challenges the notion of the "geek-gene" and suggests that it is the inherently interdependent nature of programming concepts, along with human tendency to learn at the edge of prior knowledge that is a significant contributing factor towards high attrition and failure rates (Robins, 2010). Fundamental concepts of programming imparted in first year courses are highly linked and "build upon" each other. This implies that an inability to grasp early concepts is a strong indicator of subsequent overall failure rates. We developed and introduced several strategies to our fundamental first year course based on the LEM theory. The results, although preliminary, are encouraging.

In this paper, we address one of the perennial problems of computer science– high failure and attrition rates in first year courses–and present some concrete strategies and encouraging results from our application of the LEM theory.

2 Background

COMP102: Introduction to Program Design is a first course in programming in the School of Engineering and Computer Science at Victoria University of Wellington, New Zealand. The course introduces object-oriented programming, with objects introduced fairly early in the course. The course spans one trimester (12 weeks) and introduces Java control structures, methods, parameters, top-down design, text input/output, graphical output, objects and classes, files, arrays (1D, 2D, variable sized lists), simple event-driven GUI (very constrained), and Java interfaces. We do not cover inheritance or Collection classes in this course. COMP102 is a mandatory course for all computer science and engineering majors and a popular elective for some other disciplines, such as Information Systems. In other words, COMP102 is a reasonably standard first year Computer Science/Software Engineering course. Further details of the course structure and content can be found on the course homepage: http://ecs.victoria.ac.nz/Courses/COMP102_2011T1/

3 The Problem: High Attrition and Failure Rates

Over the past 25 years, through all its minor and major modifications and variations, COMP102 has consistently exhibited high attrition and failure rates, ranging from 40 to 50%. This is a problem. Such high failure rates are common in similar courses around the world and so are the non-normal distributions of grades (especially a bi-modal distribution).

Although research into computer science education does not conclusively identify anyone or more factors that determine success or failure (Bornat, Dehnadi and Simon 2008, Cross 1970, Curtis 1984), several factors have been suggested as possible causes of high attrition and failure rates in first year programming courses. One of the most common is the notion that individuals have an innate ability to program which determines their success or failure. Determinants of this "innate programming ability", suggested over the years, include factors such as cognitive ability (verbal, mathematical, spatial, and analogical skills) (Pea and Kurland 1984, Wileman, Konvalina and Stephens 1981, Wolfe 1969) cognitive development (Piaget's stages of cognitive development and Bloom's taxonomy of educational objectives) (Bloom, Englehart, Furst, Hill and Krathwohl 1956, Piaget 1971), cognitive style (learning style, personality type etc.) (Hudak, and Anderson 1990, Myers 1995), and demographic factors (gender, age, etc.) (Woszczynski, Haddad and Zgambo 2005). In other words, most research has been focused on determining the cause assuming the problem is with “them” (i.e. the students)

4 The LEM Theory

The Learning Edge Momentum (LEM) hypothesis suggests an alternative explanation and claims that it is the human tendency to learn at the edge of prior knowledge combined with the inherently tight and highly interdependent nature of programming concepts that leads to success or failure in learning programming (Robins, 2010). In other words, since we learn at the edge of what we already know, successful acquisition of one concept makes it easier to learn other closely related concepts and vice-versa (Robins, 2010). At the heart of the LEM theory is the realization that the nature of programming is such that concepts (and constructs) “build upon” each other and failure to grasp any one component, especially in the early parts of the course, has a cascading effect – making it significantly harder to grasp later, related concepts. The LEM hypothesis is based on a simulated model of grade distributions and an extensive review of educational and psychological literature.

Our experiences suggest that programming inevitably involves dependence e.g., one cannot understand loops without understanding variables, and one cannot understand arrays without understanding loops, and so on. This highly-integrated nature of programming concepts coupled with the way people learn creates an “inherent structural bias” in first year courses leading to extreme outcomes reflected by bi-modal distribution of grades.

5 The Strategies: Improving Momentum

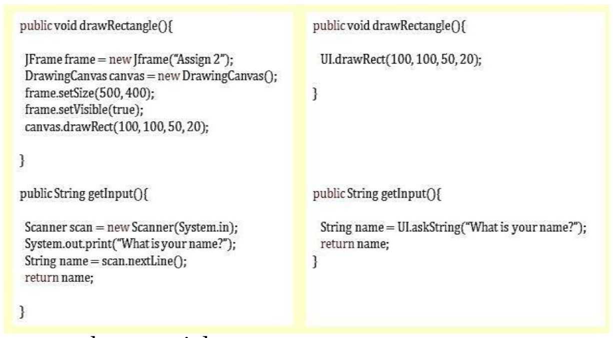
A direct recommendation of the LEM theory is for particular attention to be paid to early stages of the course ensuring everything runs smoothly and there are plenty of opportunities for grasping early concepts. Robins’ recommendations, however, were very general. To apply these recommendations, and the principles of the LEM theory, we developed a set of concrete strategies for modifying COMP102. They can be grouped into four clusters below and described in the following subsections:

- Minimizing early complexities in the course
- Minimizing dependences between early components of the course
- Maximizing chances of mastery of the early concepts and skills
- Maximizing opportunities for early recovery

In the following sections, we describe each of these.

5.1 Minimizing Early Complexity using UI Library

An ideal course from a LEM perspective would start with modules that each address a small set of concepts, skills, and knowledge, and able to be learned readily by students based on what they already knew at the beginning of the course. A typical programming course, especially in a language such as Java, has a large number of “gratuitous complexities” – concepts that are not fundamental principles of programming but are consequences of the programming language, the programming environment, or the particular details of how the lecturer has chosen to present the material.



```
public void drawRectangle(){

  JFrame frame = new Jframe("Assign 2");
  DrawingCanvas canvas = new DrawingCanvas();
  frame.setSize(500, 400);
  frame.setVisible(true);
  canvas.drawRect(100, 100, 50, 20);

}

public String getInput(){

  Scanner scan = new Scanner(System.in);
  System.out.print("What is your name?");
  String name = scan.nextLine();
  return name;

}
```

```
public void drawRectangle(){

  UI.drawRect(100, 100, 50, 20);

}


public String getInput(){

  String name = UI.askString("What is your name?");
  return name;
}
```

Fig. 1 Example of using UI library (right) to minimize early complexity

Even simple one-method programs in Java involve a lot of gratuitous complexity if they involve any input and output. For example, standard output using System.out.println involves calling a method on a static field. Even though this does not have to be explained in detail, this statement has two “dots”, in contrast to the standard pattern of <object> <dot> <method name> (<arguments>) and such inconsistencies constitute gratuitous complexity that trips up students. Standard input also includes similar complexities. The simplest form is probably to use a Scanner, but this means that for their first programs to have any input from the user, the students must deal with creating instances of a Class (and passing an argument that is a static field to the constructor), storing the object in a variable, and then calling methods on it. To use any kind of graphical output requires even more complexity. Although experience tells us that many students cope with this, the LEM theory also suggests that some students will fail the course because they got tripped up at this early stage and were unable to recover.

We designed and introduced a Java library (the “UI” library) that provides much simpler input and output, allowing students to do text input and output, and simple graphical output by calling methods on a “predefined object”. This library removed a lot of the gratuitous complexity from the early part of the course. Importantly, it made it possible to delay the construction of new objects from the second week to the third week, significantly simplifying the concepts required for the second assignment. It also allowed the construction of new objects to be introduced in a more meaningful and motivating context, rather than just as a way of getting input from the user. The library was designed to be as consistent with standard Java as possible, in order to minimize the barriers when the students have to deal with standard Java. For example, text input in the library includes methods with the same names and behaviour as the methods in the Scanner class, making it easier for students to cope with Scanner when they meet it in the context of reading data from files later in the course.

Many courses and textbooks have also introduced special libraries to reduce the complexity for new programmers. However, the UI library seems to be particularly simple to use, in comparison to the ones we have seen. More details: http://ecs.victoria.ac.nz/Courses/COMP102_2011T1/Comp102Documentation Fig. 1 shows a couple of examples of how the use of the UI library minimized complexity with the original code on the left-hand side and the same code simplified as a result of the use of the UI library on the right-hand side.

5.2 Minimizing Dependencies in Assignments

The second ideal quality from a LEM perspective is that the modules should not depend on each other, so that students can learn each module based on what they already knew at the beginning of the course, rather than having to have already mastered the previous modules. As Robbins points out, the ideal is simply not possible in programming since so many of the concepts build on top of each other – for example, parameter passing depends on understanding variables, and both conditionals and loops depend on Boolean expressions. However, since many of the early assignments had to be at least modified, if not replaced because of the new library, we were able to look again at the assignments from the perspective of minimizing dependencies. By being careful about choosing the programming tasks, we were able to significantly reduce the level of dependence between assignments 2, 3 and 4, compared to the previous year. For example, there were two pairs of programs prior to introducing LEM strategies where the second program in the pair was an extension of the first program assigned in the previous week. If a student failed in the earlier assignment, they were at an immediate and obvious disadvantage in the later assignment. In introducing LEM strategies, we eliminated all such pairs, so that each program in the first four weeks was quite different.

The changes to the library also removed some of the dependencies, so that there was no longer a dependency between the program that introduced text input and the program that was centred on creating new objects and calling methods on them (since dealing text input no longer had to introduce the concept of creating a new Scanner object).

However, there was little reduction in the dependencies between the later assignments, because they were deliberately addressing larger programs that necessarily integrated a variety of constructs and concepts from the earlier part of the course.

5.3 Maximizing Chance of Success using "Bridging Exercises"

Even though we were able to reduce some of the gratuitous dependencies between the early assignments, there were still significant dependencies, even in the first four weeks. For example, variables are introduced right at the beginning, and are used in all programs from then on; once conditionals are introduced, they are used everywhere. We believe that these dependencies are unavoidable.

Given this, it is essential to maximize the probability that students will be able to master the concepts in every one of the early assignments. This is not necessarily the same as maximizing the probability of successfully completing all the programs – all that is required to keep the momentum going is for the students to understand the new concepts in each module well enough to be able to use them and build on them in the next module.

Our previous assignments were all whole programs, and if they didn't get the program, they probably didn't get the concept either. We did not want to get rid of these "whole programs" – represent what the larger task of programming is all about and the fundamental goal of the course – but the "all or nothing" aspect is problematic according to LEM theory.

Therefore we added exercises – to enable mastery of the individual constructs and concepts, as a "bridge" into the programs which would then build on and solidify, and show their use in a realistic context. The goal of the exercises was merely mastery of individual new constructs and new concepts. Exercises were small artificial programs that were pared down to be as small as possible without being totally meaningless. Students were allowed to get as much help from tutors in the labs as they needed for completing the exercises. In order to avoid the exercises becoming a possible hindrance for the more advanced students, there were a series of exercises which were not marked and students could move to the actual (marked) program as soon as they could do 2 exercises by themselves.

5.4 Maximizing Opportunities for Early Recovery via Self-Directed e-Learning

We developed several self-directed e-learning tools to allow students maximum opportunities for revisiting materials and learning from them in a self-paced manner. These self-directed tools included video materials that were made available online to students in order to provide them with the ability to self-direct their learning. The lectures were video recorded and the recordings were made available to students online through the course homepage and in their labs. Lecture videos allowed students to view/review material in their own time at their own pace. We produced several kinds of videos: video recordings of lectures, demos of assignment programs, working review of past tests and exams, short tutorials on various topics, and additional review of lecture material. We also provided videos demonstrating the assignment programs to make sure that students understood clearly what was required.

These materials included a set of short, "YouTube-style" tutorial videos focused on single programming concepts, such as loops and methods, and working through previous exam questions. These videos were between 8 and 30 minutes. Tutorial videos took double the time to prepare as the length of the videos but were reusable from year-to-year.

5.5 Encouraging Results

We are encouraged by the preliminary results of applying the LEM theory to COMP102. Preliminary results show that the overall failure reduced to 35% (from 45% the previous year). A comparison between failure rates in before and after application of LEM theory is presented in table 1. There were 262 and 269 students in the course in each of the years respectively.

Categories	Pre-LEM	Post-LEM
Overall (of 262/269)	45%	35%
CS/ENG (of 173/169)	39%	33%
Non-CS/ENG (of 89/100)	52%	37%
Design Students (of 17/19)	75%	42%
No prior programming experience (of 127/149)	48%	42%

Table 1. Failure Rates in COMP101 Pre- and Post-Application of LEM Theory Strategies.

We conducted a course evaluation at the end of the course to gain a sense of how our strategies were perceived by the students. There were 128 responses, of which 68% indicated that they found that the exercises and lecture videos "contributed to learning"; nearly 72% said they found that the tutorial and demo videos "contributed to learning".

We also analysed the written comments on evaluation forms which favoured video resources due to their ability to help students in "revisiting concepts", "catching missed lectures", "seeing assignments work before starting on it", and easily accessing them. Similar comments were recorded for tutorial videos: "Tutorial videos helped a lot - need more of them", "tutorial videos going over last year's test helped".

We believe that if these preliminary results hold up, then there is merit in continuing with the strategies we developed and deployed in COMP102 based on the LEM theory. Further iterations of the course will provide a better indication of the sustainability of these results.

6 Conclusion

High attrition and failure rates in first year computer science and software engineering courses have traditionally been attributed to individuals' "innate" inability to program. Recent research proposed an alternative explanation in the form of the Learning Edge Momentum (LEM) theory which suggests that human tendency to learn at the edge of prior knowledge combined with the inherently tight and highly interdependent nature of programming concepts leads to success or failure in learning programming. We developed some concrete strategies in order to apply the LEM theory to our first year computer science and software engineering course and found encouraging results.

Using the strategies presented in this article – such as reducing dependencies between components and providing ample avenues for successfully grasping core concepts early on – we hope to provide everyone who attempts to learn programming a better chance at succeeding.

7 References

Bloom, B., Englehart, M.D., Furst, E.J., Hill, W.H., and Krathwohl (1956): D. Taxonomy of Educational Objectives: Handbook I Cognitive Domain, NY: Longmans

Bornat, R., Dehnadi, S. and Simon (2008): Mental models, consistency and programming aptitude, *Proceedings of the Tenth Australasian Computing Education Conference (ACE 2008)*, 53–62

Cross, E.M. (1970): The behavioral styles of computer programmers, *Proceedings of the Eighth Annual SIGCPR Conference*, 69–91

Curtis, B. (1984): Fifteen years of psychology in software engineering: Individual differences and cognitive science, *Proceedings of the 7th International Conference on Software Engineering*, 97–106

Hudak, M.A. and Anderson, D.E. (1990): Formal operations and learning style predict success in statistics and computer science courses. Teaching of Psychology, 17(4), 231–234

Myers, I.B. (1995): Gifts Differing: Understanding Personality Type, Mountain View, CA: Davies-Black Publishing

Piaget, J. (1971): The theory of stages in cognitive development, In D.R. Green, M.P. Ford, & G.B. Flamer (Eds.), Measurement and Piaget NY: McGrawHill, 1–11

Pea, R.D. and Kurland, D.M. (1984): On the Cognitive Prerequisites of Learning Computer Programming. Technical Report No.18, Bank Street College of Education, New York, NY

Robins, A (2010): "Learning edge momentum: A new account of outcomes in CS1," *Computer Science Education*, 20(1), 37-71

Wileman, S.A., Konvalina, J. and Stephens, L.J. (1981): Factors influencing success in beginning computer science courses. Journal of Educational Research, 74, 223–226

Wolfe, J.M. (1969): Testing for programming aptitude, Datamation, April 1969, 67–72

Woszczynski, A., Haddad, H. and Zgambo, A. (2005): An IS student's worst nightmare: Programming courses. *8th Annual Southern Association for Information Systems (SAIS)*, 130–133