

Integrating Programming and Systems Analysis Course Content: Resolving the Chicken-or-the-Egg Dilemma in Introductory IS Courses

Rand W. Guthrie
Computer Information Systems
California State Polytechnic University, Pomona
Pomona, CA 91768

Abstract

Most undergraduate IT programs require that students learn some computer programming as soon as possible. We have observed however, that in the subsequent systems analysis courses, students appear to have some difficulty in understanding how the design artifacts they create in their systems analysis course relate to the production of real computer programs. We believe that frequent comparisons of software design artifacts to final code improve students' ability to create good software designs. We also believe that student programming skill is directly related to software design skill. Two object-oriented systems analysis and design courses were taught at an undergraduate university covering identical concepts and content. One course however was supplemented with examples of working code that related to directly to the analysis and design examples used in the class. At the end of the two courses, the students' ability to integrate the design artifacts they learned about in class to actual code designs was evaluated through an exam that required shell code writing, reverse-engineering, and design improvement. The results indicated that students who were better programmers scored better on the evaluation exam. Students in the course that used code examples in class also performed significantly better than students in the "traditional" course. This implies that students should be taught programming first (with some high-level architectural guidance), followed by the system analysis course. Systems analysis & design courses would also benefit from using code examples that relate to analysis and design constructs.

Keywords: programming, systems analysis and design, learning styles, course integration

1. INTRODUCTION

Most undergraduate IT programs require that students learn some computer programming as soon as possible. While students seem to learn the syntax of a computer language readily enough, the quality of these early programs in terms of logic, robustness & maintainability is very weak. This often leads faculty to wonder whether we would be better off teaching students

how to design software first, before teaching them to code. Conversely, we have observed that when the systems analysis course follows the programming course, students appear to have some difficulty in understanding how the design artifacts they create in their systems analysis course relate to the production of real computer programs. Many systems analysis texts and courses that we have investigated treat the production of working computer programs very lightly if at all. Even

texts dealing with object-oriented designs and the UML, which were specifically created to address the creation of object-oriented programs, seem far removed conceptually from the world of programming in most chapters. This leaves us with a dilemma: which should come first, the programming course or the design course?

In this study we attempt to shed some light on this problem by examining student understanding of how their systems analysis artifacts relate to the production of code. We believe that programming skill is directly related to design skill. We also believe that early and frequent references that relate systems analysis concepts to final code production increase student understanding of the purpose of analysis and design processes, re-enforce learning and retention, and improve their ability to create robust designs. We test these hypotheses by comparing two courses in object-oriented systems analysis and design that cover identical material using the same textbook, but in one of the courses, we introduce and use actual code produced by the designs studied in class. Students in both courses were given an exam at the end of the course designed to test their understanding of how designs relate to actual code. The results clearly indicate that students who rate themselves as good programmers scored consistently higher than those who admitted to being less skilled in programming. The results also indicate that even students who rated themselves as being poor programmers performed better on the exam in the course where programming concepts were emphasized than those in the "traditional" course. This suggests that students should learn a programming language before the systems analysis and design course. Additionally, teaching strategies that use actual code could improve learning results in systems analysis & design courses.

2. BACKGROUND

(Booth, 2001) explains that the definition of "good learning" is evolving away from memorizing towards the development of an integrated set of skills including research, analysis, questioning and collaboration. This educational philosophy is being referred to as "Constructivism" (Gruender,

1996; Savery and Duffy 1995). In their research on the use of CASE tools in education, Fowler et. al. (2001) explains that computer science students predominantly have a learning style that is both sensory and visual, and that 80% of all students are active learners. This suggests that courses taught in a traditional fact-memorization mode may be particularly unsuited for computer science students.

Compared to traditional academic disciplines, information systems and computer science are relatively new pedagogies. These new disciplines are strongly-related to practice and therefore most courses have a high skill component. Whiddett et. al. (2000) suggest that traditional lectures do not develop skills in students. Conversely they also note that skills learned "on-the-job" are too skill-based and do not generalize well to other contexts. This suggests that university courses should be a blend of both theory and practice, rather than strongly emphasize one approach over another.

In a study involving PASCAL programming students, Fleury (1993) noted that programming students have very different "thinking habits" and motives than those of professional programmers. In particular, he notes that student tend to have a short-term perspective focused on turning in a working assignment, as compared to professionals who are far more concerned about future maintainability. This difference identifies that students are either not seeing or not being taught the larger picture in programming courses.

Perkins (1992) explains that when knowledge is "organized" and placed in a context, that the knowledge is easier to remember and more apt to be reused. Gal-Ezer and Zeldes (2000) state that "generative knowledge" as defined by Perkins preserves knowledge for a longer time, improves understanding, and is used actively.

Lebow (1993) and Savery and Duffy (1995) propose a number of teaching principles that implement constructivist pedagogy. The principles that relate to this research include:

- Provide a context for learning that supports autonomy and relatedness
- Embed the reasons for learning into the learning activity itself
- Anchor all learning into a larger task or problem
- Design an authentic task

3. RESEARCH HYPOTHESIS

Our research hypothesis is founded on the active-learning, constructivist teaching philosophies previously discussed. We believe that the students who have more programming experience are able to place their systems analysis learning more easily into a context, and are better able to conceptualize the end-result of their UML designs. This gives rise to our first hypothesis:

H1: Students who are better programmers will have a better understanding of the relationship between UML designs and final code.

Given the limited knowledge and experience of software engineering students in introductory courses, we feel that the reasons for the design (final code) should be embedded in systems analysis course content. Based on the constructivist principles of "Embedding the reasons for learning into the learning activity itself" and "anchoring" all learning into a larger task or problem," it is our expectation that the use of programming code examples in systems analysis courses will improve learning. This gives rise to our second hypothesis:

H2: Students will create better software designs in systems analysis courses that use final code examples.

4. RESEARCH METHODOLOGY

Two similar introductory object-oriented systems analysis and design courses were taught during the same term at an undergraduate university. The primary focus of the course was learning the various UML diagrams and constructs. Completion of an introductory Java programming course was a strictly enforced prerequisite. Both courses had similar gender demographics and size. Students were undergraduate

business degree majors with declared information systems emphases. Both courses covered the same material and used the same textbook (Larman 2002). The instructor teaching the traditional course was a senior member of the faculty with extensive knowledge of the subject, including the recent publication of a textbook on OO Systems Analysis & Design. The instructor teaching the integrated course was a newer member of the faculty with less experience and expertise, and had initially learned the course content by attending the senior faculty member's course two years previously. The two courses had a similar syllabus in terms of pace, exams, and projects. The instructor of the integrated course concurrently taught programming courses, the more senior faculty member had little or no recent programming experience either as a practitioner or instructor. The more experienced faculty member utilized a theoretical approach that did not emphasize any particular syntax rules or use code in any form. The instructor with current programming experience emphasized Java programming syntax in class, variable & method naming, and in method calls. Code was also used to illustrate the application of software "patterns". Students were shown actual code samples that related to UML interaction diagrams and class diagrams as part of the learning. Students were required to "reverse-engineer" simple Java programs into corresponding interaction diagrams and class diagrams. Extra credit was offered to students who completed a simple UML design project that produced a working program. These uses of code appeared in lectures, in-class activities, projects, and exams.

The students' ability to integrate programming with systems analysis & design concepts was evaluated through a one-hour exam given to the students of both courses during the last week of a regular ten-week term (three quarters per academic year). The exam was not a formal part of the course; students were offered extra credit for completing the exam on a graduated scale: the better they performed on the exam the more extra credit points they earned. Total extra credit available amounted to approximately 1/2 of 1 percentage point in the overall course grade.

The research instrument contained two parts; a survey portion and a skills portion. The survey portion asked a variety of questions to determine the student's prior programming experience, education, and skill level. Gender demographic data was also collected. The skills portion consisted of three tasks to test student's ability to integrate programming with UML design. In the first task, students were required to write a shell Java program consisting of four classes from a sequence diagram and a class diagram. Task two required student to create a class diagram by reverse engineering instructor-supplied Java source code. Task three required students to re-engineer a class diagram into "a better diagram based on your knowledge of three-tier architecture and software patterns."

The exams were evaluated by an independent teaching assistant with three terms of prior experience grading both systems analysis and Java course assignments. A total "percent correct" score was given to each exam with a moderate amount of explanatory notation included. All exams were evaluated in one session with the first group of exams graded being compared to the last group to control for familiarity bias. No significant bias was noted.

5. RESEARCH FINDINGS

A summary of the findings is shown in Table 1.

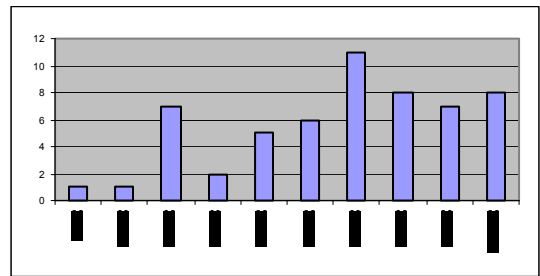
Table 1 – Summary of Scores by Group

Score Distributions	0-10%	11-20%	21-30%	31-40%	41-50%	51-60%	61-70%	71-80%	81-90%	91-100%	Average
Overall	1	1	7	2	5	6	11	8	7	8	0.593
Traditional Course	1	1	5	2	3	5	7	3	2	2	0.502
Integrated Course	0	0	2	0	2	1	4	5	5	6	0.706
Poor Programmers	1	0	2	1	3	4	3	2	2	0	0.501
Okay Programmers	0	0	2	0	2	1	4	5	5	6	0.592
Good Programmers	0	0	0	0	1	0	2	2	1	3	0.720

The average score for the entire group was .593. Scores showed a tendency towards a normal distribution with a marked skew towards 100% (Figure 1). Student performance on the evaluation exam ranged broadly, with about half the students (23/56) scoring above 70%

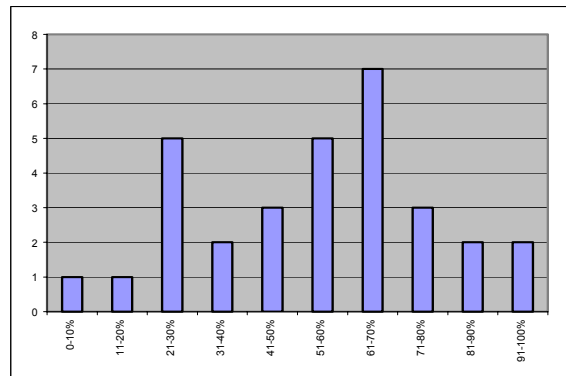
(23/56) scoring above 70% which would be considered a "passing" grade in most courses. This figure is surprisingly low, given that this performance level could be a fair predictor of how well students really understood the material they were supposed to learn.

Figure 1: Distribution of overall student evaluation scores



The distribution of scores for the class taught "traditionally" (without the use of computer code) shows an approximate normal distribution of grades, with 20 out of 31 scoring between 31% - 80%. Two students scored in the 0%-20% range, and four students scored in the 80% - 100% range. Twelve students received scores less than 50%.

Figure 2: Distribution of Tradition Methodology Group Scores



The distribution of scores for the class taught with the use of computer program code integrated with systems analysis & design concepts are significantly skewed to the right, with 20 out of 25 scoring between 60% and 100%. Four students received scores less than 50%, and no students received scores less than 20%

Figure 3: Distribution of Integrated Methodology Group Scores

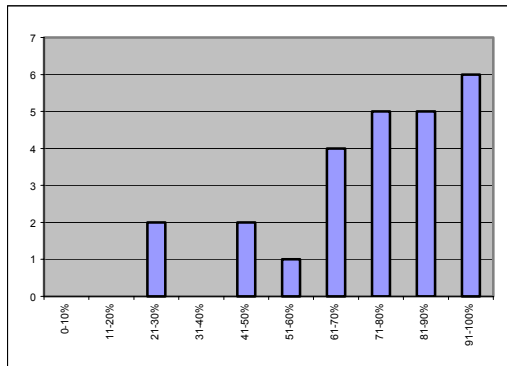


Figure 5: Distribution of "Okay" Programmer Scores

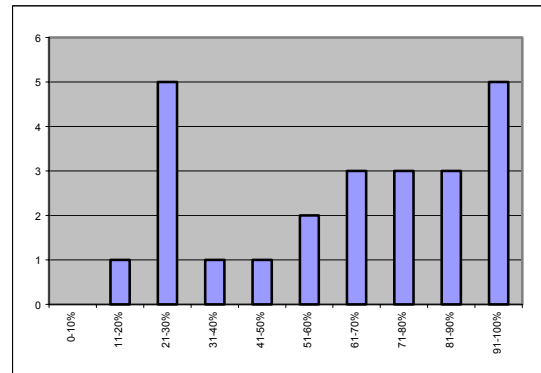


Figure 4: Distribution of "Poor" Programmer Scores

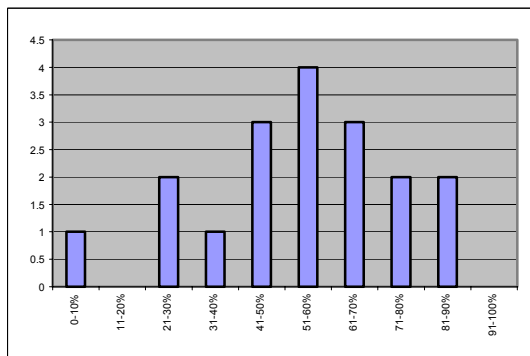
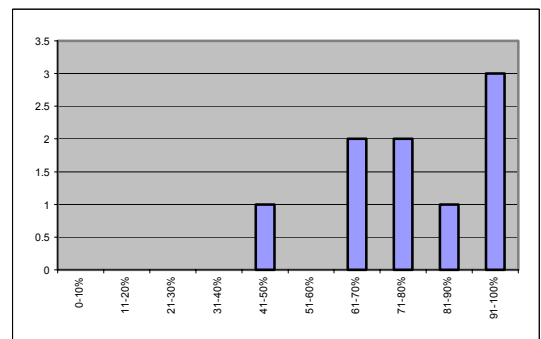


Figure 6: Distribution of "Good" Programmer Scores



Almost half the students surveyed rated themselves as "okay" programmers (24 out of 57). The distribution of "okay" programmers between the two courses was even: 12/12. Eighteen students reported themselves as either being "poor" or "not so good". The distribution between the two courses was 10 in the traditional course and 8 in the integrated course. Nine out of the fifty-seven said they were "pretty-good" or "excellent", with the distribution as 4 in the traditional course and 5 in the integrated course. None classified themselves as "hackers", and five students in the traditional course did not answer that question. Students who rated themselves as "poor" or "not so good" averaged 0.501 with a normal distribution over most of the range (Figure 4). Students who said they were "okay" programmers averaged 0.592 with the distribution being skewed to the right except of a group in the 20% - 30% range (Figure 5). Students who rated themselves as "pretty-good" or "excellent" averaged .720 with a right bias (Figure 6).

6. DISCUSSION OF RESULTS

Both hypotheses were supported by the research findings. It is clear that students who reported being better at programming performed better as a group on the evaluation than those who reported being less skilled in programming. It is interesting to note that students in general tended to be modest in evaluating their programming ability. Since the identity of those taking the survey and their scores are known, in the future we would like to compare self-report performance to student grades in prior coursework. It is also clear that those students who attended the "integrated" course that used code examples scored higher as a group. This implies that students integrate software engineering principles better when the relationship between the results of their designs are emphasized throughout the course. We believe that these results support widely used "folk pedagogies" (Booth 2001) that introduce students to software engineering with

a programming course followed by a design course. Given that program design has a huge impact on robustness and maintainability, we do not suggest however, that introductory programming courses should ignore teaching the basics of good design. Concepts such as three-tier architecture, separation of concerns, and iterative development are basic ideas that students can readily understand, yet provide a foundation of good design practice right from the start. This study suggests therefore that the "chicken" should indeed come before the egg, but with proper course content, the chickens will be matured and ready to be productive "egg-layers" in the follow-on systems analysis and design courses.

7. REFERENCES

- Booth, S. (2001). "Learning Computer Science and Engineering in Context." *Computer Science Education*, 11(3), 169-188.
- Fleury, A. E. (1993). "Students' beliefs about Pascal Programming." *Journal of Educational Computing Research*, 9(3), 355-371.
- Fowler, L., J. Armarego, and M. Allen (2001). "CASE Tools: Constructivism and its Application to Learning and Usability of Software Engineering Tools." *Computer Science Education*, 11(3), 261-272.
- Gal-Ezer, J and A. Zeldes (2000). "Teaching Software Designing Skill." *Computer Science Education*, 10(1), 25-38.
- Gruender, C. D. (1996). "Constructivism and learning: A philosophical appraisal." *Educational Technology*, 36, 21-29.
- Larman, C. (2002). *Applying UML and Patterns* (2nd ed.). Upper Saddle River: Prentice-Hall.
- Lebow, D. (1993). "Constructivist values for instructional systems design: Five principles toward a new mindset." *Educational Technology Research and Development*, 41, 4-16.
- Perkins, D. N. (1992). *Smart schools: from training memories to educating minds*. New York: Free Press.
- Savery, J. R. and T.M. Duffy (1995). "Problem Based Learning: An Instructional Model and Its Constructivist Framework." *Educational Technology*, 35(5), 31-38.
- Whiddett, R.J., B.X. Jackson, and J.A. Handy (2000). "Teaching Information Systems Management Skills: Using Integrated Projects and Case Studies." *Computer Science Education*, 10(2), 165-177.