Examining the Number of Concepts Students Apply in the Exam Solutions of an Introductory Programming Course

Pratibha Menon
menon@pennwest.edu
Department of Computer Science and Information Systems
Pennsylvania Western University, California
California, PA, 15419, USA

Abstract

Instruction in an introductory programming course is typically designed to introduce new concepts and to review and integrate the more recent concepts with what was previously learned in the course. Therefore, most exam questions in an introductory programming course require students to write lines of code that contain syntactic elements corresponding to the programming concepts covered during the instruction. This study investigates the number of concepts involved in the exam problems of an introductory Java programming course. In addition, this study compares how the increase in the number of concepts correlates with the ability of students to write error-free lines of code. The instructional method adopted in this study focuses on providing students with a problem-solving schema and a resultant programming plan that integrates many concepts to meet the problem's goal. Results from this study indicate that as the course progresses through the semester, students, on average, apply appropriate problem-solving schemas and programming plans to produce more error-free lines of code, despite an increase in the concept count in the problems. Furthermore, the exam problems later in the course repeat the application of cluster concepts that have appeared in the past exam. This paper illustrates how programming is a cumulative skill and that repeating and building upon the applications of these concept clusters several times through the course increases the likelihood that students will produce more correct lines of code as the semester progresses.

Keywords: Concepts, Introductory-Programming, Lines-of-code, Exams, Program.

1. INTRODUCTION

Failure rates in introductory programming courses have prompted several researchers to identify the causes that make these courses difficult for students (Watson & Li, 2014; Medeiros et al., 2019; Bennedsen & Caspersen, 2019). One thread of research has explored the types of assessment used in introductory programming courses to determine the factors that make the test items difficult (Zur & Vilner, 2014; Ford & Venema, 2010). Exams in a programming course typically consist of tasks designed to assess how well students can apply various programming constructs to solve problems. The difficulty of a programming task in an exam item may be evaluated subjectively from student self-assessments or objectively by collecting data about the problems' characteristics and their solutions (Braarud, 2001).

Prior studies have found that the difficulty of exam questions in introductory programming courses taught by different instructors depended on various measures of complexity such as the degree of explicitness, reference to an external and unfamiliar domain, hard-to-learn concepts, linguistic complexity, intellectual complexity level based on Bloom's taxonomy. (Sheard et al., 2011; Sheard et al., 2013; Harland, D'Souza & Hamilton, 2013). In these studies, the complexity

measures were evaluated based on the instructors' perceptions, and the exam questions' resulting difficulty was inferred through the students' marks.

Studies also noted that exam questions in introductory courses were predominantly composed of integrative code-writing questions that required students to apply multiple concepts (Petersen et al., 2011). Evaluating questions' content and cognitive requirements indicate that students must internalize a large amount of introductory programming content and gain enough practice solving problems to succeed in the exams. While academics can evaluate the difficulty of the exams, they tend to underestimate the total number of concepts used versus those evaluated (Simon et al., 2012). For example, a question on loops requires students to master basic concepts such as Boolean logic, variables, data types, operators, and the associated syntax before constructing a loop that solves a given problem. Some programming concepts are so fundamental that they are used in every code-writing instance and must be committed to long-term memory.

Cognitive load theory explains that concepts not fully internalized must be reasoned in the working memory (Ericsson & Kintsch, 1995; Berssanette & de Francisco, 2022). In addition, the mind is limited in its ability to work with multiple concepts simultaneously, and therefore, students who need more mastery of fundamental concepts face an increasing cognitive burden (Muller et al., 2007). Results from prior studies argue that we may be asking students in introductory programming courses to master too many concepts in a short time (Goldman et al., 2010).

This study investigates the intrinsic cognitive load of course contents in an introductory programming course by quantifying the conceptual complexity of the exam problems used to assess student learning. The conceptual complexity of an exam problem is measured by the number of distinct concepts contained in an optimal code solution. Instruction in an introductory programming course takes place by introducing students to new concepts and integrating them with the previously taught concepts. Therefore, exam questions are formulated to assess the conceptual knowledge gained by students by evaluating how well students learn to integrate and apply these concepts to solve problems.

This study investigates how the cognitive load introduced in learning programming concepts impacts the ability of students to produce correct code. First, the conceptual complexity of the course contents is measured by counting the concepts expected to be applied in each of the solutions to the exam problems. Then, the learning outcome is measured as the correctness of the lines of code of the solutions produced by students at three different points through a fifteen-week semester.

The approach of evaluating the conceptual complexity of exam problems by identifying and counting the number of concepts applied in the expected solution, as explained in this paper, could be used by instructors to objectively gauge the difficulty of exam questions in their introductory programming courses. This paper also examines the possibility of using lines of code as a reasonably good metric within an introductory programming course to score the exam solutions for a student's ability to write code by applying the required concepts. The research question of this study is formulated as follows:

RQ1: How do the number of distinct concepts students apply to solve exam problems increase as the semester progresses in an introductory programming course?

RQ2: How do the number of concepts students apply to solve a programming problem correlate with their ability to produce an error-free solution as they obtain instruction and practice to solve application-based problems in an introductory programming course?

This study takes place in a live classroom with class lectures and a detailed code walkthrough demonstrating the instructor's practices in applying the concepts to solve problems. The introductory programming course investigated in this study has three monthly exams that test the ability of students to recall, analyze and apply their conceptual knowledge to write code sequences. The findings of this study have implications for designing instruction that supports instructors and students in taming the complexity of integrating many concepts in programming solutions.

2. THE COURSE CONCEPTS AND LINES OF CODE

Course exams are a valuable proxy for deriving curricular expectations and determining what instructors understand as essential. This study explores the concepts used in the exam questions of an introductory Java programming course. While no standard concept inventory exists for

introductory programming courses, researchers have used varying methods to derive a list of essential concepts. For example, Tew and Guzdial have used the contents of textbooks to identify ten critical topics (Tew & Guzdial, 2010). Schulte and Bennedsen shortlisted 28 topics from the literature and asked instructors to rate the difficulty and relevance of each (Schulte & Bennedsen, 2006). A prior study by Petersen et al. evaluated exam contents and observed that the number of concepts considered by instructors while evaluating and grading programming solutions in an exam is fewer than those used to construct the program (Petersen et al., 2011).

This study draws the concepts from a concept inventory created for CS1 courses (Goldman et al., 2010), as depicted in Table 1.

Concept	
Program Structure	Arithmetic Operators
Method Structure	Assignment Operator
Method Parameter	Operator Precedence
Method Return	Proper use of parenthesis
Method Call	Expression
Syntax	Statement
Data Types	Conditionals
Variables	Decision Structures
Literals	Loops
Boolean Operators	Nested Loops
Variable Scope	String methods

Table 1. List of Concepts

All the concepts displayed in Table 1 are treated as being equally difficult, although studies have qualitatively identified certain threshold concepts that are more critical than others in the learning process (Meyer & Land, 2005; Sanders & McCartney, 2016). However, the results of these studies could only broadly identify the threshold concepts, for example, as pointers or object-oriented programming. Both these topics were not included in the curriculum of the introductory course, whose content is investigated in this paper. A study by Cherenkova et al. (2014) investigated a large dataset of CS1 code-writing attempts and found that certain straightforward application of concepts tend to be problematic even towards the end of the term.

Thus far in computing education, evaluating code complexity of solutions has either involved expert evaluation or the use of convenient metrics, such as the number of syntactic elements in a piece of code. While metrics-driven software engineering has fallen out of favor, they are a convenient quantitative method for measuring code (DeFranco & Voas, 2022). One popular metric, lines of code, is commonly used to measure developers' productivity. Lines of code is also an intuitive metric for measuring software size since its effect can be visualized. For example, lines of code could be used to count a program's volume of instructions (or statements). However, not all lines of code in a Java program may terminate in a semi-colon. For example, a for-loop does not contain a semi-colon but forms a line of code containing an executable entity.

Lines of code may be composed differently by novice and professional programmers (Kramer et al., 2017). Skilled developers can apply more syntactic entities with far less code. However, most novice developers, such as the students who attend an introductory programming course, only pack a physical line of code with a few logical constructs. While learning, it is easier for novices to comprehend and write code if each physical line contains the application of fewer logical constructs and the program is written in a step-by-step manner, as a logical sequence, using separate lines.

This study considers two types of heuristics - the number of concepts expected to be applied in an optimal solution and the expected lines of code to study how these metrics could infer the problem's complexity. The lines of code count all the instances of using syntactic elements that correspond to the concepts used to solve the problem. On the other hand, the concept count only considers the "distinct concepts" used to formulate an optimal solution. It is important to note that the number of distinct concepts applied in the solution should be optimal, which means these concepts are the ones whose application is necessary and cannot be avoided in the solution. This study explores how these two heuristics - the number of distinct concepts expected in an optimal solution and the total number of lines of code in the students' solution, could gauge students' learning progress to solve increasingly more significant problems in a course during a semester.

3. THE STUDY

This study takes place in a 15-week introductory Java Programming class in an undergraduate-

level Computer Information Systems program at a public university. The course has three-unit exams that are spread out throughout the course. Exam1 covers the topics of decision structures. Exam 2 focuses on loops, and Exam 3 tests the ability of students to modularize their code using methods.

The Exams

The exams comprised coding problems that required students to apply their conceptual knowledge. Some questions only require students to analyze code. Most questions, however, require students to write a code solution. The exam questions were of variable points, and scores were assigned to each question based on the correctness of the code lines expected in the solution. In addition, students are given partial points to a solution based on the percentage of the number of lines of code answered correctly, compared to the lines of code expected in a correct solution. Given the stringent time allotted to complete the exams, no open-ended questions could have resulted in a high degree of code variability in the solutions. Each hour-long, closed-book exam was conducted in a classroom, and the exam was strictly timed and proctored. Students access and submit their exams through the course learning management system. Furthermore, due to the time limits of the exams, students were not asked to use a compiler to run their solutions during the exam. The exam's primary intent was to test students' ability to recall the syntax and apply their conceptual knowledge to write java program statements.

Points carried by an exam question correlated with the number of lines of code students had to write or analyze. For example, short answer questions required students to write or analyze one or two lines of code. Medium-sized questions had solutions that contained between 5-11 lines of code. A more extensive solution had about 12-26 lines of code. A summary of the characteristics of the exam questions for the three exams is given in Table 2. The upcoming sections of this paper will illustrate how the conceptual complexity of the exam questions evolves between the three exams. It must be emphasized that the exam questions were created such that the program solutions resembled a multi-step problem solving process, where each step involves application of a different cluster of concepts. Therefore, care was taken to ensure that larger code sizes in the exam solutions did not just result from repetition of similar statements involving the same group of concepts.

	Exam1	Exam2	Exam3
Duration	1 hour	1 hour	1 hour
Max points	50	70	100
# of questions	8	8	5
Points/ question	between 5 and 20	between 5 and 20	between 10 and 50
Approx # of expected Lines of Code / question	between 1 and 16	between 1 and 16	between 7 and 25

Table 2 – Exam summary

The Instruction

Before each exam, students were exposed to the exam topics via class lectures, code demonstrations, and weekly assignment exercises. The course contents are covered in four modules. Appendix D shows the assignment problems from each module along with the key concepts covered in that module. Through these learning activities, students are exposed to various problems that apply the concepts listed in Table 1. Appendix D also categorizes problem into various types such as calculators, checkers, counters etc. The code demonstrations used to instruct problem solving methods in class, covered several application scenarios and code development techniques. Every code walkthrough thoroughly explained a programming problem and solutions using a program plan that reflected the instructor's problem-solving schema. Appendix C also shows how a simple problem could be broken down into various steps to develop a code walkthrough. Appendix C also shows a flow chart used by the instructor to plan the code walk through for any given problem.

The assignment problems provided means for students to solidify their conceptual understanding and apply (or modify) their problem-solving schemas to solve similar problems from a different context. Appendix D shows a sequence of assignment problems that also allows the reuse the concepts covered in the previous assignments. The assignment problems are similar in scope and scale to the ones whose solutions are explained in the code demonstrations. The exams help the instructor evaluate how correctly students transfer the problem-solving schemas and program plans involving multiple concepts to fit the specific context of an exam problem.

Exam solutions of 25 students who attended all three exams were collected. Any information identifying a student was removed from the solutions. Student submissions were not matched across the exams. Students' answers were scored for correctness by comparing them with the expected statements and syntax of the lines of code in the instructor's solutions. Every line of code that formed a statement was checked for correctness and assigned a point only if there were no errors.

4. RESULTS

Concepts in the Exam Questions

An analysis of the exam questions by the course instructor revealed all the concepts that a student needs to apply to solve each exam question. Figure 1 shows that the total number of concepts increased in the later exams. Figure 1 names each exam problem using the exam number and the problem number. For example, E3P5 stands for Exam 3, problem 5. Appendix A shows the mapping of each exam problem to the distinct concepts that need to be applied to write an optimal solution. Appendix B lists a partial list of questions from the three exams.

Figure 1 also shows the approximate number of lines of code students were expected to write or analyze in each exam problem. It can be observed from Figure 1 that even a single line of code could contain syntactic elements that represented multiple concepts. For example, problem E1P1 (described in Appendix B) required students to analyze a statement that contained a compound Boolean expression containing comparison and logical operators. While students were evaluated based on their understanding of Boolean expressions, they also needed to understand several foundational concepts, such as operator precedence, proper use of parenthesis, and the Java syntax used to comprehend a Boolean expression.

In Exam 1, problems 1, 2, and 3 (listed as E1P1, E1P2, and E1P3) required students to analyze a given statement, and problems 4, 5, 6, 7, and 8 (depicted as E1P4 – E1P8) required students to write lines of code using the concepts required to write if-else or switch statements.
Appendix B describes some of the questions from Exam 1. For example, writing lines of code that contain decision structures and the actions that follow the truth value of each conditional expression in the decision structure brings together 10 – 14 concepts, as observed in the concept mapping table in Appendix A. The bars corresponding to E1P5, E1P6, E1P7, and E1P8 in Figure 1 also show the many concepts used to solve these problems.

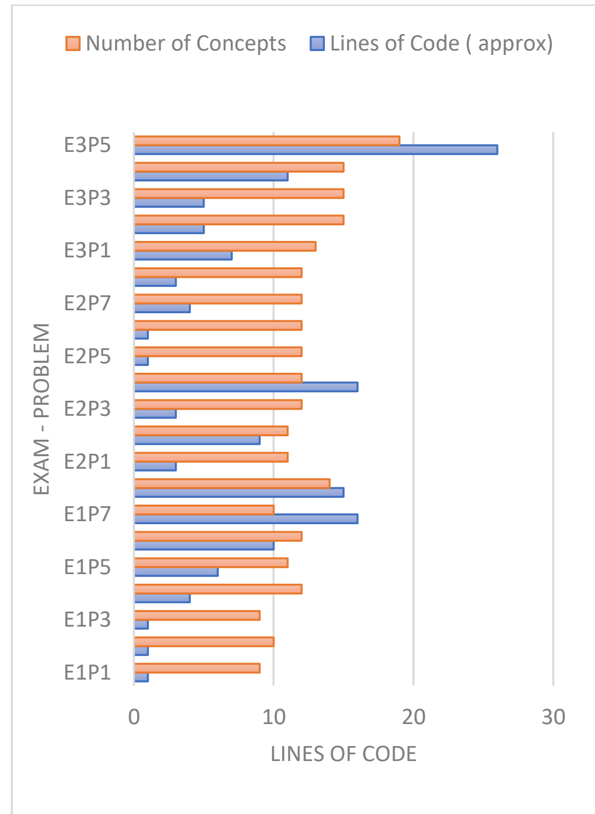


Figure 1. Concept count and the expected lines of code for exam questions.

Exam 2 required students to know how to write applications that use while, do-while, and for loops. Programs that included loops also contained foundational concepts such as variables, Boolean expressions, different types of operators, and simple conditional statements. To apply loops in a program, students also need knowledge of data types and syntax rules to compose expressions and statements. The number of lines of code in Exam 2 composed of problems E2P1 till E2P8 are shown in Figure 1. Even though two problems may have the exact concept count, their lines of code may differ based on the program plan for the solution. Some problems may have additional statements requiring using a different set of operators and print statements, thereby adding the number of lines of code without increasing the concept count. For example, this was the case in problem E2P4 compared to other problems with similar concept counts.

Appendix A shows the concept mapping of these problems, and Appendix B describes a partial list of the problem statements. The concept count of

the Exam 2 problems is relatively high compared to the expected lines of code in the solutions. For example, writing a simple for-loop requires knowledge of arithmetic and Boolean operators, appropriate use of variables and their scope, and the syntactic elements used to compose expressions and statements. Additional concepts are used to write statements that form the loop's actions.

Exam 3 requires students to write modular code using methods. Students must comprehend the questions and translate the requirements into code by writing the correct return type, arguments, and statements in the method's body. Depending on the problem's requirements, a method's body may require the application of concepts such as arithmetic and Boolean operations, decision structures, or loops. Therefore, questions in Exam 3 also included the concepts that constituted the problems in exams 1 and 2. Appendix A reveals some of the concepts involved in Exam 1 and 2 that were repeated in Exam 3. Figure 1 shows that Exam 3 questions E3P1, E3P2, E3P3, and E3P4 have code lines with high concept counts. The question E3P5, which carried the most points and concept count, required students to write a menu-driven application that repeated several if-else statements to direct the program based on user choices during execution time. Repeating the if-else statements added code lines that used the same concepts, and therefore, the concept count did not increase as much as the code lines did. The table in Appendix B describes problem E3P5.

The Pearson correlation results indicated a significant positive relationship between the lines of code per exam problem and the number of concepts ($r = .592$, $p = .005$). Therefore, for this study, the number of correct lines of code written by students could be used to gauge their latent conceptual knowledge. It is important to ascertain this positive correlation if the correct number of lines of code is to be used to measure student performance in the exams.

Student Performance

Student submissions were scored based on the percentage of the expected lines of code that were correct for each question. Tables 4, 5, and 6 show the average values of correctly written lines of code (or statements) for each question from the three exams. The tables also show the average percentage score per problem and the number of concepts in each question. The Pearson correlation results indicated a significant positive relationship between the percentage of correct lines of code for each solution and the number of concepts used to solve the problem ($r = .67$, $p < .001$).

Problem	Avg Score	# Concepts	Avg Correct lines of code
E1P1	71.41%	9	0.714
E1P2	85.71 %	10	0.857
E1P3	52.38 %	9	0.524
E1P4	68.57 %	12	2.744
E1P5	96.03 %	11	5.76
E1P6	75.24 %	12	7.52
E1P7	58.57 %	10	9.376
E1P8	90.71 %	14	13.605
Avg values	75.00%	10.875	5.1375

Table 4 Exam 1 Results

Problem	Avg Score	# Concepts	Avg Correct Lines of Code
E2P1	82.86 %	11	2.487
E2P2	76.19 %	11	6.858
E2P3	73.33 %	12	2.199
E2P4	75.24 %	12	12.032
E2P5	77.14 %	12	0.771
E2P6	91.43 %	12	0.914
E2P7	86.67 %	12	3.468
E2P8	88.1 %	12	2.643
Avg Values	79%	11.75	3.9215

Table 5 Exam 2 Results

Problem	Avg Score	# Concepts	Avg Correct Lines of Code
E3P1	86.39 %	13	6.048
E3P2	73.47 %	15	3.675
E3P3	85.71 %	15	4.285
E3P4	81.63 %	15	8.976
E3P5	87.66 %	19	22.802
Avg values	82%	15.4	9.1572

Table 6 Exam 3 Results

Tables 4, 5 and 6 indicate that problems that required students to apply more concepts were the ones that students tended to score the most. In addition, the tables show that the average test score percentage increased after every exam. If

the number of concepts used in a problem indicates its complexity, these results mean students are getting better at handling many concepts as they progress through the course.

Looking at the mapping of concepts in Appendix A, one can observe that Exam 1 uses many foundational concepts that are re-applied in all subsequent exams. Students also revisit many of these concepts in the assignment problems that follow Exam 1. Therefore, many code lines in the second and third exams repeat the concepts used in the first exam. Appendix A also shows that many of the same concept cluster together in various exam problems. For example, almost all programs use variables, data types, operators, and expressions. Knowledge of the correct syntax to construct statements is fundamental to all problems. Introducing newer concepts, such as decision structures and loops, helps to reinforce the use of foundational concepts covered earlier in the course, such as Boolean expressions and the use of different types of operators. The use of basic operators and inputs and outputs methods reoccur in almost all the programs that involve decision structures and loops. Therefore, repeated application of these concept clusters to meet the problem's sub-goals and create a more extensive program plan allows students to write correct code involving these concepts in subsequent exams.

A solid understanding of basic concepts and how they occur as a cluster to meet the program's goal and sub-goals allows students to incrementally integrate newer concepts successfully as they learn to write more extensive and complex programs.

5. DISCUSSION

It may appear concerning that students must grasp as many as 11 concepts by Exam 1, conducted during week 6 of the course. The Table in Appendices A and B shows that even writing a simple statement to solve a Boolean or Arithmetic operation requires knowledge of a cluster composed of many concepts. For example, nine concepts in the first three questions of Exam 1(E1P1, E1P2, and E1P3) are written using a single line of code that applies operator precedence rules. Even though there is no drastic increase in the number of concepts elsewhere in the course, students must learn to integrate newer concepts with what they already know to write programs as the course progresses.

Results from Tables 4, 5, and 6 indicate that students could write more correct lines of code as the semester progressed. Therefore, reapplying the same concept cluster many times throughout the semester and the familiarity gained would have led to mastery and better performance later in the course. However, it is to be noted that learning to solve different application problems happens not just by repeated exposure to the application of concepts but through the dynamic process of reconfiguring prior concepts to integrate a new concept required to solve a problem. Therefore, instruction could be designed to support acquiring new conceptual knowledge by reconfiguring and reapplying prior knowledge and skills and learning to apply a newer concept.

The progression of code writing exercise problems could play an essential role in helping students learn how to restructure their problem-solving schema and recombine prior concepts to solve new problems. For example, as evident from the assignment problem types in Appendix D, basic arithmetic operators, covered early in the semester, could be applied to develop various types of calculators. However, in the later module, problems that incorporate checkers into calculators will require students to incorporate Boolean operators into their pre-existing arithmetic operators and expressions schema. Problems that students solve later in the course require them further incorporate basic knowledge of arithmetic and Boolean operators in new ways to implement decision structures and loops. While there is considerable repetition of concept clusters throughout the course, there is also a need to restructure the previously learned cluster of concepts in new ways to solve different types of problems.

The problem-solving schema transferred through instructional code walkthrough helps students re-configure and re-apply concept clusters to solve problems. The instructional code walkthrough could help students identify the goals and sub-goals of the problem and then identify and configure a concept cluster to meet the sub-goals. For example, Appendix C shows the goals and sub-goal identification for a simple PIN identification problem, one of the assignment problems given to students. This solution pattern for the PIN identification problem could be modified to create applications that may validate user inputs or allow users to log in with a username and password. This problem and solution pattern could also be extended using loops to incorporate reattempts to check the user inputs.

Classic works in learning theory have argued that learners accumulate schema, or a problem-

solving plan, rather than build their solution from scratch by applying all the elementary concepts (Rist, 1989; Clancy & Linn, 1999). Per this model, errors in applying a schema to solve a problem in an unfamiliar context or modifying the schema to fit the problem requirements may reveal flaws in the learner's understanding of the concepts used to compose the solution. The application of schema theory to computing pedagogy has taken a renewed interest, as indicated by the data-mining efforts to study common error patterns encountered by students during their learning process (Zehetmeir et al., 2016).

Repeated schema application or its modification to the problems' contexts allowed students to re-apply a cluster of concepts and assemble their solutions multiple times throughout the course. For example, based on the instructor's report, decision structure problems E1P5, E1P6, and E1P8 were analogous to problems in previously graded assignments. However, problems E1P1, E1P2, and E1P3 were single-line problems that did not directly resemble any assignment problems or were applied as part of a larger program plan. Even though students would have used smaller Boolean expressions to build decision structures, problems E1P1, E1P2, and E1P3 needed students to reason about the solution by considering every concept in the statement. Problem E1P7 was another problem that required a considerable modification of the assignment schema. Similar results were observed in Exam 2, where students scored the most if they could successfully identify similar problems from instructional code walkthroughs and assignments and transfer the schema to solve the exam problem.

Students scored the most in Exam 3 because the code inside the bodies of the methods repeated and reconfigured several code schemas previously covered in the assignments and exams. For example, the problems in the final exam required students to apply loop or decision structures in the body of the methods. Students could successfully write the body of the methods in Exam 3 if they learned how to integrate the method concepts with the problem-solving schemas used to solve loops or decision structure problems earlier in the course. A solid application of schemas within the body of a method allowed them to score partial points for a problem, even if they made mistakes directly related to the concept of a method, such as writing the method header or providing a correct return statement.

6. CONCLUSIONS

The result of this study indicates that programming is a cumulative skill and that as the semester progresses, students learn to write conceptually complex lines of code by accumulating and integrating many concepts into their solutions. The course starts with a high initial number of concepts and progresses with a relatively gradual increase of newer concepts that must be integrated with previously learned concepts. Integrating newer concepts to solve application problems also provides means to reconfigure code patterns and master the base concept clusters applied earlier in the course. Instructional code walks through, and practice assignments support the acquisition of programming skills by repeatedly integrating newer concepts into a cluster of concepts that appear in past assignments and exams.

This study confirms a positive correlation between the error-free lines of code produced for a solution and the number of concepts that students need to integrate to produce a solution. An explanation of why students can write correct lines of code despite increasing the conceptual complexity of the solutions is that they can learn the problem-solving schema and apply code patterns involving concept clusters. Students and instructors cope with an extensive concept count by clustering the concepts into code patterns corresponding to a problem schema. This study's finding has implications for designing instructional activities to help students recognize the instructor's problem-solving schema that deals with clusters of concepts that could be re-configured to meet a goal. Students may then remember each concept in isolation due to its meaningful association with other concepts in a solution's code pattern.

This study primarily focused on the ability of students to solve exam problems like those used during the instructional process. Future studies could investigate the complexity of exam problems by characterizing the concept clusters that appear in various problem-solving schemas. In addition, studies could be conducted to learn how students transfer problem-solving schema to unfamiliar problems. Finally, the difficulty of exam problems could be assessed based on not just the concept count but also conditioned on prior exposure to similar problems.

7. REFERENCES

Bennedsen, J., & Caspersen, M. E. (2019). Failure rates in introductory programming. *ACM Inroads*, *10*(2), 30–36. https://doi.org/10.1145/3324888.

Berssanette, J. H., & de Francisco, A. C. (2022). Cognitive Load Theory in the Context of Teaching and Learning Computer Programming: A Systematic Literature Review. *IEEE Transactions on Education*, *65*(3), 440–449. https://doi.org/10.1109/te.2021.3127215.

Braarud, P. I. (2001). Subjective Task Complexity and Subjective Workload: Criterion Validity for Complex Team Tasks. *International Journal of Cognitive Ergonomics*, *5*(3), 261–273. https://doi.org/10.1207/s15327566ijce0503_7.

Cherenkova, Y., Zingaro, D., & Petersen, A. (2014). Identifying challenging CS1 concepts in a large problem dataset. *Technical Symposium on Computer Science Education*. https://doi.org/10.1145/2538862.2538966.

Clancy, M. J., & Linn, M. C. (1999). Patterns and pedagogy. *The Proceedings of the Thirtieth SIGCSE Technical Symposium on Computer Science Education*. https://doi.org/10.1145/299649.299673.

DeFranco, J. F., & Voas, J. (2022). Revisiting Software Metrology. *Computer*, *55*(6), 12–14. https://doi.org/10.1109/mc.2022.3146648.

Ericsson, K. A., & Kintsch, W. (1995). Long-term working memory. *Psychological Review*, *102*(2), 211–245. https://doi.org/10.1037/0033-295x.102.2.211.

Ford, M., & Venema, S. (2010). Assessing the Success of an Introductory Programming Course. *Journal of Information Technology Education: Research*, *9*, 133–145. https://doi.org/10.28945/1182.

Goldman, K., Gross, P., Heeren, C., Herman, G. L., Kaczmarczyk, L., Loui, M. C., & Zilles, C. (2010). Setting the Scope of Concept Inventories for Introductory Computing Subjects. *ACM Transactions on Computing Education*, *10*(2), 1–29. https://doi.org/10.1145/1789934.1789935.

Harland, J., D'Souza, D., & Hamilton, M. (2013). A Comparative Analysis of Results on Programming Exams. *Proceedings of the Fifteenth Australasian Computing Education Conference (ACE2013).*

Kramer, M., Barkmin, M., Tobinski, D., & Brinda, T. (2017). Understanding the Differences Between Novice and Expert Programmers in Memorizing Source Code. *IFIP Advances in Information and Communication Technology*, 630–639. https://doi.org/10.1007/978-3-319-74310-3_63.

Meyer, J.H.F. & Land, R. Threshold concepts and troublesome knowledge (2005): Epistemological considerations and a conceptual framework for teaching and learning. *High Educ 49, 373–388.*

Medeiros, R. P., Ramalho, G. L., & Falcao, T. P. (2019). A Systematic Literature Review on Teaching and Learning Introductory Programming in Higher Education. *IEEE Transactions on Education*, *62*(2), 77–90. https://doi.org/10.1109/te.2018.2864133.

Muller, O., Ginat, D., & Haberman, B. (2007). Pattern-oriented instruction and its influence on problem decomposition and solution construction. *Proceedings of the 12th Annual SIGCSE Conference on Innovation and Technology in Computer Science Education*. https://doi.org/10.1145/1268784.1268830.

Petersen, A., Craig, M., & Zingaro, D. (2011). Reviewing CS1 exam question content. *Proceedings of the 42nd ACM Technical Symposium on Computer Science Education*. https://doi.org/10.1145/1953163.1953340.

Rist, R. S. (1989). Schema creation in programming. *Cognitive Science*, *13*(3), 389–414. https://doi.org/10.1016/0364-0213(89)90018-9.

Sanders, K., & McCartney, R. (2016). Threshold concepts in computing. *Proceedings of the 16th Koli Calling International Conference on Computing Education Research*. https://doi.org/10.1145/2999541.2999546.

Schulte, C., & Bennedsen, J. (2006). What do teachers teach in introductory programming? *Proceedings of the Second International Workshop on Computing Education Research*. https://doi.org/10.1145/1151588.1151593.

Sheard, J., Simon, Carbone, A., Chinn, D., Laakso, M. J., Clear, T., de Raadt, M., D'Souza, D., Harland, J., Lister, R., Philpott, A., & Warburton, G. (2011). Exploring programming assessment instruments. *Proceedings of the Seventh International Workshop on Computing Education Research*. https://doi.org/10.1145/2016911.2016920.

Sheard, J., Simon, Carbone, A., Chinn, D., Clear,

T., Corney, M., D'Souza, D., Fenwick, J., Harland, J., Laakso, M-J., & Teague, D. (2013). How difficult are exams? A framework for assessing the complexity of introductory programming exams. In A. Carbone, & J. Whalley (Eds.), *Proceedings of the Fifteenth Australasian Computing Education Conference (ACE 2013): Adelaide, Australia, 29 January - 1 February 2013* (Vol. 136, pp. 145 - 154). (Conferences in Research and Practice in Information Technology (CRPIT); Vol. 136). Australian Computer Society Inc. http://crpit.com/confpapers/CRPITV136Sheard.pdf

Simon, D'Souza, D., Sheard, J., Harland, J., Carbone, A., & Laakso, M. J. (2012). Can computing academics assess the difficulty of programming examination questions? *Proceedings of the 12th Koli Calling International Conference on Computing Education Research - Koli Calling '12*. https://doi.org/10.1145/2401796.2401822.

Tew, A. E., & Guzdial, M. (2010). Developing a validated assessment of fundamental CS1 concepts. *Proceedings of the 41st ACM Technical Symposium on Computer Science Education*. https://doi.org/10.1145/1734263.1734297.

Watson, C., & Li, F. W. (2014). Failure rates in introductory programming revisited. *Proceedings of the 2014 Conference on Innovation &Amp; Technology in Computer Science Education - ITiCSE '14*. https://doi.org/10.1145/2591708.2591749.

Zur, E., & Vilner, T. (2014). Assessing the assessment-Insights into CS1 exams. *2014 IEEE Frontiers in Education Conference (FIE) Proceedings*. https://doi.org/10.1109/fie.2014.7044330.

Appendix A
Exam problem Concepts

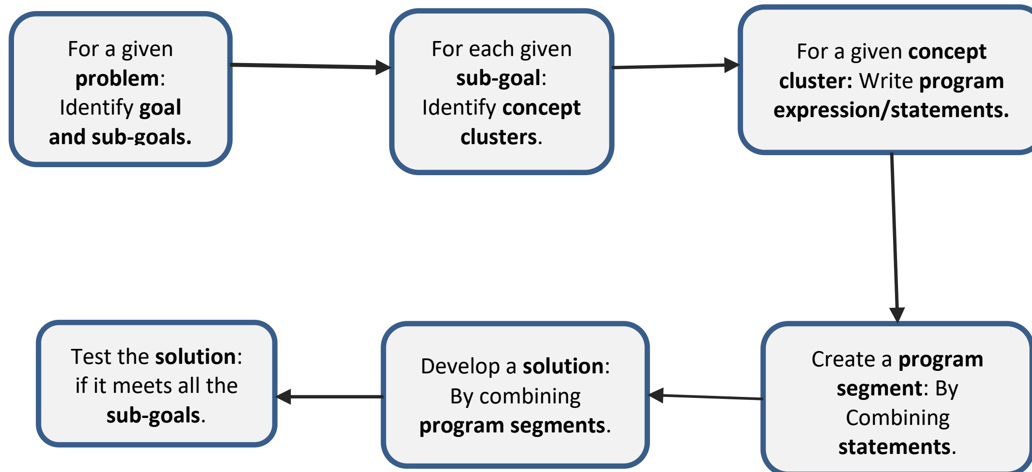
	Exam 1								Exam 2								Exam 3				
Concepts	**P 1**	**P 2**	**P 3**	**P 4**	**P 5**	**P 6**	**P 7**	**P 8**	**P 1**	**P 2**	**P 3**	**P 4**	**P 5**	**P 6**	**P 7**	**P 8**	**P 1**	**P 2**	**P 3**	**P 4**	**P 5**
Method Structure																	x	x	x	x	x
Method Parameter																	x	x	x	x	x
Method Return																	x	x	x	x	x
Method Call																					x
Syntax	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x
Data Types	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x
Variables	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x
Literals	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x
Boolean Operators	x	x	x	x	x	x		x	x	x	x	x	x	x	x	x	x	x	x	x	x
Arithmetic Operators		x			x	x		x	x	x	x		x	x	x	x	x				x
Assignment Operator	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x
Operator Precedence	x	x	x	x		x		x													x
Expression	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x
Statements	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x
Conditionals				x	x	x	x	x			x			x				x	x	x	x
Decision Structures					x	x	x	x											x	x	x
Loops									x	x	x	x	x	x	x	x					
Nested Loops																x	X	x			
String methods				x								x						x			
Variable Scope													x	x	x			x	x	x	x
Print Methods							x	x	x		x	x	x		x	x			x		x
Input(Scanner) Methods				x				x		x		x						x		x	x

Concepts used in each exam problem for the three exams. The grey cells indicate the main **course concept/topic that is evaluated in the problems. Cell that are marked with an 'x'** indicate the concept that is used in the lines of code of a correct solution.

Appendix B
Some of the Exam Problems from Exam 1, 2 and 3

<table>
<tr><th>Exam Problem</th><th>Problem/ Question</th></tr>
<tr><td>E1P1</td><td>What will be the value of boolean var1, which is given as : var1 = (((a*b)<=5)&&(b<1)); if you substitute a = 2 and b = 1 ?</td></tr>
<tr><td>E1P3</td><td>What will be the value of boolean var1, which is given as : var1 = (a.equals("Apple")); if you substitute a = "apple"</td></tr>
<tr><td>E1P5</td><td>Given two input variables : double length and double width. Given one output variable : double perimeter.
Assume that values of length and width are already obtained. Write if statement (just the if statement(s) with the action performed , not the entire program) for the following conditions:
Check if the dimensions are big as follows: If length is greater than 20.0 or width is greater than 20.0, give an output to tell the user that the dimensions are too big.
Check if the dimensions are small as follows: if the length is less than 5.0 or width is less than 5.0, give an output to tell the user that the dimensions are too small.
If the dimensions are neither too big or too small, based on the above two checks - tell the user that the dimensions are within the proper range of values . Then, calculate the area = length * width and print the value of that perimeter.</td></tr>
<tr><td>E1P8</td><td>Write a program that obtains from the user the age of a child in months. The program will determine the required next vaccinations based on the given age. Write a complete program that will look at the given age in months and determine the next vaccination required. You may skip commenting your code for this exam to save time. You program should compile and be logically and syntactically correct.
<table>
<tr><th>Baby's Age in months</th><th>Next Vaccination</th></tr>
<tr><td>0</td><td>HepatitisB</td></tr>
<tr><td>Greater than 0, less than 6 (inclusive)</td><td>DaTP</td></tr>
<tr><td>Greater than 6, less than 12 (inclusive)</td><td>MMR</td></tr>
<tr><td>Greater 12</td><td>"Will complete the program later on"</td></tr>
</table></td></tr>
<tr><td>E2P2</td><td>In this problem you will write a loop in which you ask users to enter the price of an item and add that price to the total price. In your loop, you will ask the user to enter a 1 to “Scan” and a 2 to “quit”. Loop until the user types 2 - to quit.
For as many times, as the user enters a 1- to "Scan an item' : ask the user for the price of the items , and add this price to a variable called totalPrice.
The variables price and totalPrice are both doubles .Assume that the Scanner object is already declared and named as input.
Declare any extra variables that you have used in your loop - other than price, totalPrice or input.</td></tr>
<tr><td>E2P6</td><td>The for loop shown below loops several times and produces a final value of i that is used to calculate the value of j. However, there is an error : int j = i % 5 ; //This statement shows an error : "cannot find symbol i"
Errored code:
for(int i = 1; i<100; i=i*5){
System.out.println (i);
}
Rewrite the code above so that it fixes the error given in the error statement</td></tr>
<tr><td>E3P3</td><td>Define/Write a method called codeThePlayer that takes two argument – an integer called playerID and a String called playerName . This method has a void return. If the playerID value is equal to 100, the method prints out the following : "Admin ID ". Else, if the playerID is not a 1, the method prints out the playerName, followed by the statement: "Not Admin".</td></tr>
<tr><td>E3P5</td><td>Menu driven program
Write a program called pointsCalculator that calculates the total points earned by using a credit card for travel and hotel stays. The program provides the user with the following menu :
“ Enter 1 to select mileage points”
“Enter 2 to select a hotel points”
“Enter 99 to quit”
If the user enters a 1 , ask the user to enter the mileage (which will be a double type). Scan the mileage and call a method called calculateMileage that takes in as mileage as a parameter. This method returns a double value to be stored in a variable called mileagePoints. Print out the value of mileagePoints.
If the user enters a 2, ask the user to enter the number of hotel stays(which will be an integer type). Scan the hotel stays and pass this variable as a parameter to the method calculateHotelPrice . This method returns a double variable called hotelPoints. Print out the values of hotelPoints.
Assume the methods are already defined - you just need to call them in the code. You also don't need to implement a while loop.</td></tr>
</table>

Appendix C
Problem-Solving Steps/Schema Breakdown Used for Instructional Code Walk Through



Problem	Checker - Simple PIN validation
Statement	Write a program that validates a user based on a 4-digit PIN value. The program allows the user to input their 4-digit PIN and compares this value with the value on file. Assume that the PIN value stored in the file has been obtained and stored in variable in your program.
Goal	Validate the user's PIN value with the PIN value on file and take appropriate actions for pasisng and failing validation.
Sub-goal 1	Input: Program should obtain the PIN value from the user.
Sub-goal 2	Compare for equality : Program should compare the PIN value with the PIN value on file.
Sub-goal 3	Action 1: Output an validation message if the PIN values matches / validation passes.
Sub-goal 4	Action 2: Output an error message if the PIN values do not match/ validation fails.
Sub-goal 1 program segment	a. Declare a variable to store the user PIN - [Data type,variable,assignment operator,initial value/int literal,expression,statement] : int userPIN = 0; b. Prompt the user for user PIN value - [Data type,String literal] : "Enter the 4-digit user PIN " c. Write a print statement- [String literal,method,statement]: System.out.println("Enter the 4-digit user PIN "); d. Scan and save the user input from the console - [Scanner method,data type,assignment operator,variable,datatype, expression,statement]: userPIN =
Sub-goal 2 program segment	a. Declare file PIN variable with a value of 6788 and save the value in that variable - [Data type,variable,assignment operator,expression,statement]: int filePIN = 6788; b. Boolean Expression for comparison -[Comparison operator, data types,variables, expression] : userPIN == filePIN c. If statement using comparison operator: [Decision structure, comparison operator, data types, variables] : if(userPIN == filePIN){..} else {...}
Sub-goal 3 Program segment	a. Create a validation message- [Data type/String literal] : "You are validated." b.For the if condition print the validation message- [String literal, method, statement] : if(userPIN == filePIN) {System.out.println("You are validated.");}
Sub-goal 4 program segment	a. Create an error message- [Data type/String literal] : "PIN Error, PIN not valid." b. For the else condition print the validation message- [String literal, method, statement] : else { {System.out.println("PIN Error, PIN not valid.");}

APPENDIX D
Assignment Problems Categorized by Problem Types and Concept Clusters

List of Application Problem Patterns in Assignments	
Module 0- Variable, Data types, Scanner methods	**Calculators - Data types, Arithmetic Expressions, Input/Output, Strings**
	Shipping Cost
	Taco Price
	TypeCasting Inputs
	Flooring Cost
	HealthData
	Make Change
	Checkers: Data types, Boolean expressions, Input/Output, Strings, IF/ELSE
Module 1 - Decision Structures	Find Special Values : filtering out special values from a series of input data
	Range Checker - identifying the range of a given input value
	Age Checker - identifying the age group that a person falls under
	PIN validation - validating user PIN value (without retries)
	Checkers +Calculators: Data types, Boolean/Arithmetic Expressions, Input/Output, IF/ELSE (mutliple ifs, else if)
	Score Difference - figuring out game winners based on score differences
	Age Checker - Binning for creating Histograms
	Age & ZipCode checker - Demographic categorization problem
	Year To Century Converter
	Ticket Price Based on age /product type / discount code
	Electrice Power Consumption Calculator for multiple home appliance types.
	Decisions&Policies: Boolean/ Arithmetic Expressions, Input/Output, nested IF/ELSE, Strings
	Rock Paper Scissor Game
	Labor Charge Calulator for Lawn Service
Module 2 - Loops	**Counters: Data types Boolean/Arithmetic Expression, Input/Outputs, Strings, Loops (while and for)**
	Shopping Data Input till user wants to quit using sentinel value
	PIN Validation with retries
	Interest Calculator
	DivideByTwo series generator
	ABCounter ForLoop Implementation
	ABCounter WhileLoop Implementation
	FutureTuition Calculator with inflation rates
	InsectGrowth - series generator with varying parameters
	Loops with Decisions - Data types,Boolean and Arithmetic expression, Input/Output, Strings, IF/ELSE , Loops
	Validating Inputs - with infinite retries.
	Menu Driven Application with multiple rounds of entry
	Password, Username validation with re-tries and lockout
Module 3- Methods	**Methods - Return types, parameters, Data types,Boolean, Arithmetic expressions, Input/Output, Strings, IF/ELSE, Loops,**
	Print Shapes : Methods to print different shapes without parameters
	Customaisable Face Printer with parameters - Methods to print different types of faces, with parameters
	Dinner Price Calculator1 - Methods for user input, entrée price, discount calculator
	eBayFee: Methods for identifying user types, fee calculation, output
	Length Convertors : Methods for multiple types of conversions
	Ticketing Application 1: Methods for user input, calculation
	Ticketing Application 2 - Methods for user input, decision making, calculation
	Dinner Price Calculator2 - Menu driven app, methods for user input, entrée price , drink price, receipt
	TaxApplication - Methods for: user inputs, output display, calculator and category checkers