2014

# Transition from Concepts to Practical Skills in Computer Programming Courses: Factor and Cluster Analysis

Candido Cabo
*CUNY New York City College of Technology*

# Transition from Concepts to Practical Skills in Computer Programming Courses: Factor and Cluster Analysis

**Dr. Candido Cabo, New York City College of Technology/CUNY**

Candido Cabo earned the degree of Ingeniero Superior de Telecomunicacion from the Universidad Politecnica de Madrid in 1982, and a Ph.D. in Biomedical Engineering from Duke University in 1992. He was a post-doctoral fellow at Upstate Medical Center, State University of New York, and a research scientist in the Department of Pharmacology at the College of Physicians and Surgeons of Columbia University. In 2000, he joined New York City College of Technology, City University of New York (CUNY) where he is a Professor in the Department of Computer Systems Technology. Since 2005, he has been a member of the doctoral faculty at the CUNY Graduate Center. His research interests include computer science and engineering education and the use of computational models to understand and solve problems in biology.

# Transition from Concepts to Practical Skills in Computer Programming Courses: Factor and Cluster Analysis

**Abstract**

Computer programming courses are gateway courses with low passing grades, which may result in student attrition and transfers out of engineering and computer science degrees. Barriers to success include a good understanding of programming concepts and the ability to apply those concepts to write viable computer programs.

In this paper, we analyze the determinants of the transition from concepts to skills in computer programming courses using factor and cluster analysis. The purpose of this study is to answer the following questions related to computer programming teaching and learning: 1) Which are the correlations and interdependencies in student understanding of different computer programming concepts?; 2) Which are the cognitive challenges that students find when learning programming concepts?; 3) How the understanding of different programming concepts relate to practical skills in computer programming; 4) What determines a successful transition from understanding the concepts to the ability to write viable computer programs?

After several computer programming concept assessments in this first Java Programming course, we grouped the students' performance into seven different categories: assignment operators, repetition structures, selection structures, program design using methods, arrays, classes and Java syntax. Factor analysis identified two factors (components) grouping the interdependencies and correlations between programming concept categories. The first component correlated with the repetition and selection categories, and could be referred to as the "algorithmic" component. The second component correlated with the methods, arrays and assignment categories, and could be referred as the "structural" component. Student performance in conceptual categories related to the "algorithmic" factor was significantly better than in conceptual categories related to the "structural" factor. Cluster analysis showed that student performance in the "structural" conceptual component is predictive of the student's ability to solve practical computer programming problems.

We conclude that a strong emphasis in the structural components of computer programming (i.e. program design using methods, use of the assignment operator, and use of data structures like arrays) is necessary for a successful transition from concepts to skills in computer programming courses.

## 1. Introduction

First year problem-solving and/or computer programming courses are gateway courses with low passing rates, which may account for student attrition and transfers out of computer science degrees. A number of challenges have been identified over the years by the computer science education community. It has been shown that an understanding of the problem domain to be solved by implementing a computer program should be a prerequisite for writing the computer program itself [1, 3, 14, 16]. Students' inability to create a mental model [8] of a given problem domain hinders their ability to develop problem-solving skills and write computer programs.

Another difficulty encountered by novice programmers is the syntax of computer programming languages, which is often overwhelming to students who get distracted from solving problems by the obscurity of the statements and program organization. This difficulty was recognized early in computer programming education, and different strategies including graphical languages and animations of program states were developed [17]. One approach to increase success in first-year programming courses is a shift from teaching programming to teaching problem-solving skills [4, 5, 9]. This approach has been successful and avoids some of the problems that hinder progress in the development of thinking skills that are important for computer programming. However, this approach has also been criticized because the translation of a problem solution to a computer program is not obvious [14, 18]. The challenges faced by students and educators in learning and teaching computer programming have been summarized in a recent review [15].

Following earlier findings in computer education research we require our students to take a problem-solving course before their first programming course. It has been showed that introducing narrative elements in pre-programming problem-solving courses (a pedagogical approach that has been called programming narratives) is more effective than traditional approaches using a full-fledge programming language as a tool to help students develop computer programming problem-solving skills [10, 11]. To facilitate the implementation of programming narratives we currently use *Alice* (www.alice.org), a programming environment that allows learners to create interactive animations while learning computer programming concepts. However, despite the benefit of using programming narratives to help students develop problem-solving skills, the transition from pre-programming problem-solving courses to courses where students should master a full-fledge programming language remains a challenge [14, 18].

Two barriers to success in computer programming courses include a good understanding of programming concepts and the ability to apply those concepts to write viable computer programs. The purpose of this study is to answer the following questions related to computer programming teaching and learning: 1) Which are the correlations and interdependencies in student understanding of different computer programming concepts?; 2) Which are the cognitive challenges that students find when learning programming concepts?; 3) How the understanding of different programming concepts relate to practical skills in computer programming; 4) What determines a successful transition from understanding the concepts to the ability to write viable computer programs?

## 2. Methods

### 2.1 Participants

Our institution is one of most racially, ethnically, and culturally diverse institutions of higher education in the northeast: 31.5% of students are African American, 33.8% Latinos, 20% Asian/Pacific Islander, 11.3% Caucasian, and 0.6% Native Americans. At project initiation, the College spring 2013 enrollment was 16,208.

We report data from performance assessments from 22 students who took a Programming Fundamentals course in spring 2013. In this course, students use Java as the programming language of choice to help develop their conceptual and practical programming skills. For all students, this is the first programming course in their curriculum. However, before this course, all students had taken a Problem-Solving course in which they used pseudocode, flowcharting and Alice (www.alice.org) to learn basic procedural and object-oriented programming concepts. The goal of the Problem-Solving course is to teach programming concepts without the burden of learning a full-fledge programing language. However, basic Java programming is introduced in the last three weeks of the Problem Solving course to facilitate the transition to the Programming Fundamentals course.

## 2.2 Exploratory Factor Analysis

After several computer programming concepts assessments in the first Java programming course, we grouped students' performance into seven different categories: assignment, repetition (for/while structures), selection (if/else structures), methods, arrays, classes and general syntax. Student performance in concepts and skills was assessed at three different times during the semester.

Exploratory factor analysis is a data reduction technique that aims at finding hidden correlations and interdependencies between different variables and grouping them in a number of overarching factors or components. Estimating the number of factors is tricky, and therefore to estimate the number of factors in the factor analysis we used different criteria. We used SPSS to extract the number of factors using the Kaiser-Guttman [6] (number of eigenvalues greater than one) and Cattell's scree test [2]. We also used FACTOR [12] to estimate the number of factors using Horn's parallel analysis [7]. The Kaiser-Meyer-Olkin measure of sampling adequacy was 0.674, above the suggested minimum of 0.5. Interpretation of the extracted factors can be made easier by orthogonal factor rotation. We used the varimax rotation method with Kaiser normalization.

## 2.3 Cluster Analysis

After the factors or components underlying the different conceptual categories have been identified, it is possible to derive scores for each student on each factor. We used hierarchical cluster analysis, using the Euclidian distance as a proximity measurement, to classify students' factor scores and to group students in different clusters reflecting their responses to conceptual assessments. The number of clusters was determined by inspection of the dendrogram, a display representing visually the distances at which clusters are combined.

## 3. Results

### 3.1 Student Performance in Programming Concepts and Practical Skills

Figure 1 shows the individual performance of students in concepts and skills assessments (range 0-100). Performance in concepts can be mapped to the first two levels of Bloom's taxonomy learning structure (knowledge and comprehension level), and performance in skills can be mapped to levels three and four (application and analysis level). We considered 70% (equivalent to a C grade) an acceptable ("passing") performance in the assessments (vertical and horizontal dashed lines in Figure 1). Forty-one percent of the students did not have an acceptable performance either in concepts or in skills. (For reasons that will become clear later, we represent those students in Figure 1 with a solid circle.) Thirty-six percent of the students had an acceptable performance both in concepts and skills (students represented in Figure 1 with an unfilled circle). Twenty-three percent of students had an acceptable performance in concept assessments but not in practical skill assessments (students represented in Figure 1 with crossed squares).

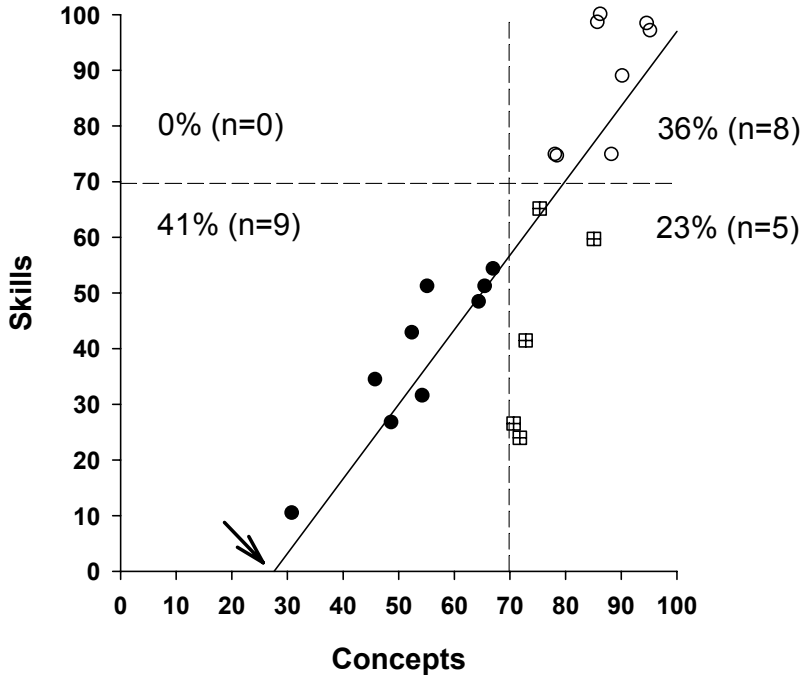


Figure 1. Overall student performance in programming concepts and skills (n = 22). Dashed lines mark an acceptable performance in computer programming concepts and skills assessments (70%). Unfilled circles represent students with acceptable performance in concepts and skills; crossed squares represent students with acceptable performance in concepts but not in skills; solid circles represent students with poor performance in concepts and skills. The solid line is the regression line (Skills = 1.34 *Concepts – 37, with $r^2 = 0.71$). Arrow shows the intercept of the regression line with the x-axis.

Figure 1 illustrates the two barriers faced by novice computer programming students: a good understanding of programming concepts and the ability to apply those concepts to write viable computer programs. About 59% (13 out of 22) of students had an acceptable understanding of programming concepts. And about 61% (8 out 13) of the students with a good understanding of programming concepts were able to transfer that knowledge into practical programming skills.

Our analysis of Figure 1 relies in a somehow arbitrary threshold that marks the difference between satisfactory and poor performance in concepts and skills (70%, dashed lines in Figure 1). However, an analysis of the regression line (which is not dependent in arbitrary thresholds) can lead us to similar conclusions. Linear regression (solid line in Figure 1, $r^2 = 0.71$) indicates that performance in programming concepts is correlated with performance in practical programming skills (see also Figure 2 below). Also, most students perform better in concepts than in practical skills (shown by the positive intercept of the regression line with the concept-axis; arrow in Figure 1), indicating a barrier in students' ability to apply concepts to the solutions of practical computer programming problems.

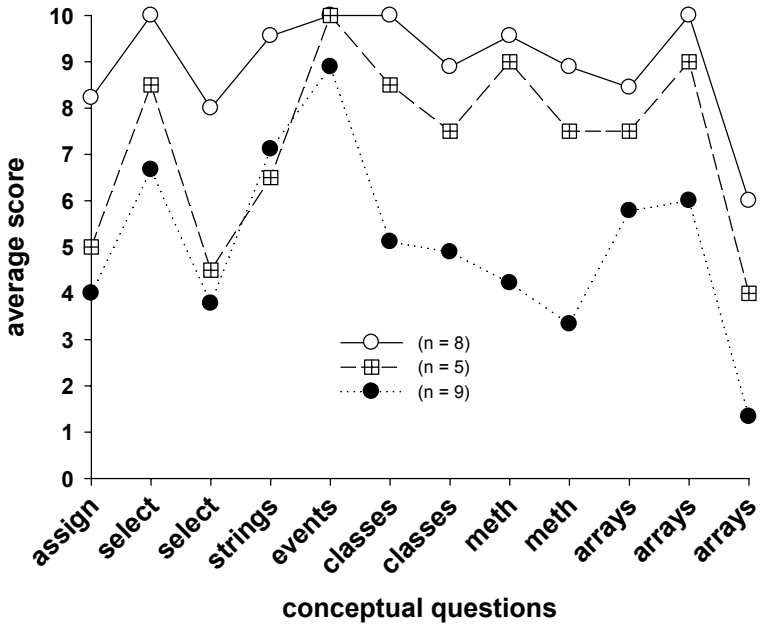


Figure 2. Average scores (range 0-10) in an assessment of computer programming concepts for students with acceptable performance in both concepts and skills (unfilled circles), for students with an acceptable performance in concepts but not in skills (crossed squares), and for students with poor performance in both concepts and skills (solid circles). Symbols are the same as in Figure 1.

There were no students with an acceptable performance in skills but poor performance in concepts (Figure 1). This suggests that without a good grounding in the understanding of the

concepts, it is very unlikely to develop acceptable practical programming skills. But, is there an acceptable (minimum) level of conceptual understanding to be able to develop acceptable practical programming skills? Figure 2 shows that students having an acceptable performance in both concepts and skills (unfilled circles) have a better overall performance in concepts than the other two groups (crossed squares, solid circles). This suggests that there might be a minimum level of conceptual understanding that is necessary in order to succeed in the development of practical programming skills. On the other hand, Figure 2 shows that the performance in the three groups is similar in some concepts (for example, events), while in other concepts the performance is markedly different (for example, methods and arrays). So, which are the more important concepts (or group of concepts) that students should master to develop acceptable practical programming skills? Is the understanding of all concepts equally important for the development of practical programming skills?

### 3.2 Exploratory Factor Analysis

To further understand the nature of students' understanding of computer programming concepts, and the hidden correlations and interdependencies between programming concepts in those seven different categories, we performed exploratory factor analysis.

We grouped student performance in computer programming concepts assessments in seven different categories: assignment, repetition, selection, methods, arrays, classes and syntax. Factor analysis identified two factors or components grouping the interdependencies and correlations between programming concept categories. Figure 3 shows a plot of the factor loadings in the orthogonally rotated space with iso-loading factor lines, which illustrate the percent of correlation of the different conceptual categories with a given factor or component.

Factor loadings are the correlations between the different categories and the extracted factors (components), and therefore their value is between +1 and -1. The first component correlated with the repetition (correlation 0.9) selection (correlation 0.83), and classes (correlation 0.61) categories, and, since it involves concepts necessary to implement computer algorithms, it could be referred to as the "algorithmic" component. The second component correlated with the methods (correlation 0.89), arrays (correlation 0.81) and assignment (correlation 0.64) categories, and, since it involves concepts on data structures, data assignment and program organization, it could be referred to as the "structural" component. The correlation of the syntax category with any of the factors was $< 0.60$.

Student performance in concept categories related to the "algorithmic" factor was significantly better (paired t-test, $p < 0.05$) than student performance in concept categories related to the "structural" factor (Figure 4). This finding suggests that students have more difficulty with the structural components of computer programming which, therefore, need more attention and emphasis in the classroom.

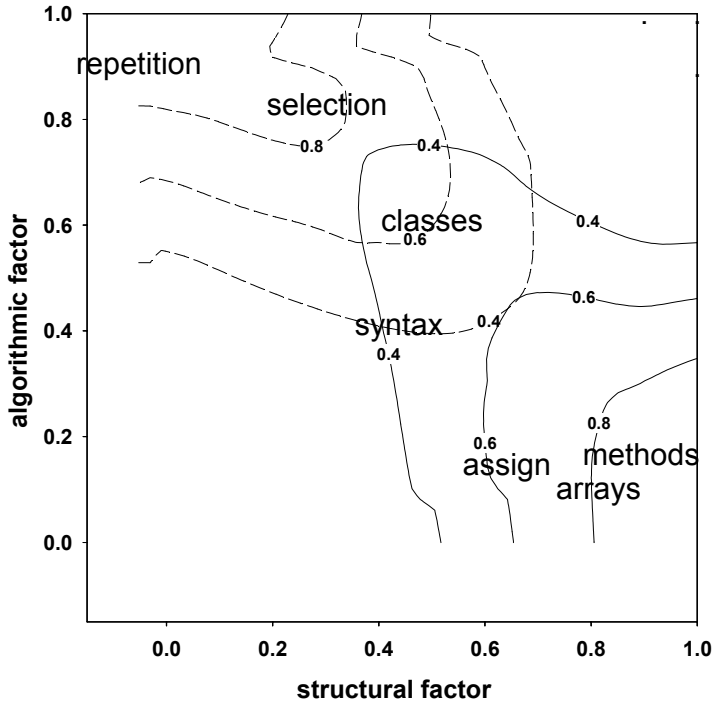


Figure 3. Factor plot illustrating the correlations of the seven conceptual categories with the "algorithmic" and "structural" factors. Solid iso-correlation lines show the correlation of the different conceptual categories with the "structural" factor. Dashed iso-correlation lines show the correlation of the different conceptual categories with the "algorithmic" factor. For clarity, only three isolines (0.4, 0.6 and 0.8) are shown.

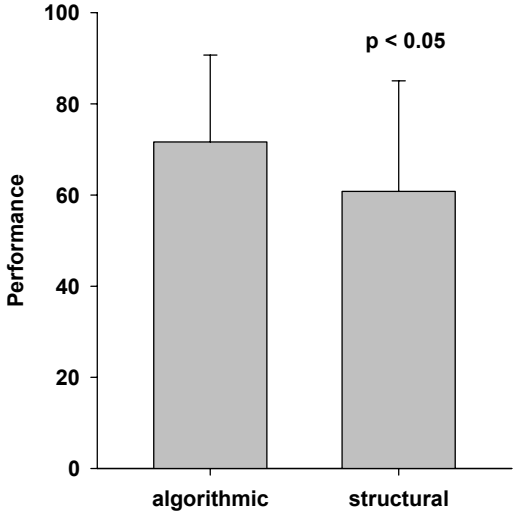


Figure 4. Performance in concept categories (range 0-100) related to the "algorithmic" factor (repetition, selection and classes) and in concept categories related to the "structural" factor (assignment, methods and arrays).

3.2 Cluster Analysis

After the factors or components underlying the different conceptual categories have been identified, it is possible to derive scores for each student on each factor. Figure 5 shows a plot of factor scores on the "algorithmic" and "structural" factors for all 22 students. Hierarchical cluster analysis of the factor scores indicated that students could be grouped in three clusters (dashed lines separate the different clusters in Figure 5). Students in the same cluster are similar with respect to their factor scores and are dissimilar to students in other clusters. Note that the clustering is mostly determined by the factor scores on the "structural" factor (Figure 5). For example, students with a factor score on the "structural" factor > 1 belong to cluster #3, regardless of the factor score on the "algorithmic" factor. Likewise, students with a factor score on the "structural" factor < 0, belong to cluster #1, regardless of the factor score on the "algorithmic" factor. For each cluster there is a similar range of variation for the factor scores on the "algorithmic" factor.

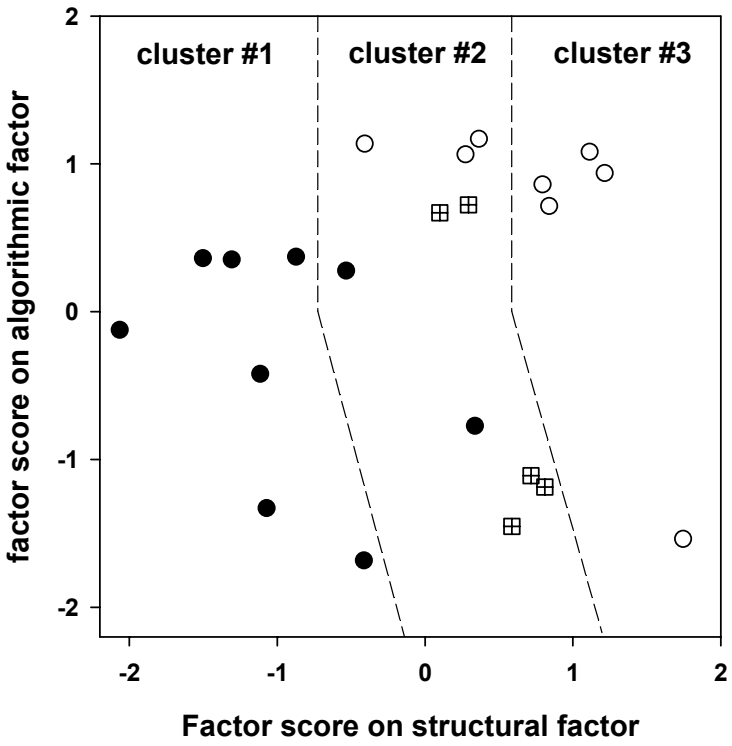


Figure 5. Plot of factor scores for all 22 students. The dashed lines separate the different clusters formed by hierarchical cluster analysis based on the Euclidian distances of the factor scores. The symbols representing the different students (solid circles, unfilled circles, crossed squares) are the same as in Figure 1.

It is instructive to compare Figure 5 to Figure 1. Note that, students belonging to cluster #1 in Figure 5 overlap considerably (7 out of 9, or 78%) with students with a poor performance in both

concepts and practical skills (solid circles in Figure 1). Also note that, students belonging to cluster #3 in Figure 5 overlap considerably with (5 out of 8, or 62%) with students with a satisfactory performance in both programming concepts and skills (unfilled circles in Figure 1). In comparing the grouping of students in Figures 5 (using cluster analysis of factor scores on concept assessments) and in Figure 1 (using performance in both concepts and skills assessments), it should be kept in mind, that the boundaries used in Figure 1 (70% performance, dashed lines) are somehow arbitrary. A different acceptable performance could have been chosen.

Overall, these results suggest that student clustering based on the factor scores on the "algorithmic" and "structural" factors, which are measurements based only on performance in *conceptual* assessments is a good predictor of student performance in *practical* programming skills. Since the clustering is mostly determined by the factor scores on the "structural" factor, we can conclude that student performance in the "structural" conceptual component is predictive of the student's ability to solve practical computer programming problems. This is also consistent with the results in Figure 4 which indicate that students have more difficulty with concepts related with the structural factors of computer programming than with concepts related to algorithmic factors.

## 4. **Discussion**

Using factor analysis we have identified two factors or components grouping the interdependencies and correlations in student understanding of programming concept categories. The first component correlated with the repetition and selection categories, and could be referred to as the "algorithmic" component. The second component correlated with the methods, arrays and assignment categories, and could be referred as the "structural" component. Student performance in conceptual categories related to the "algorithmic" factor was significantly better than in conceptual categories related to the "structural" factor. Cluster analysis showed that student performance in the "structural" conceptual component is predictive of the student's ability to solve practical computer programming problems.

Others before us have identified different factors that are important for student success in computer programming. Earlier work emphasizes the distinction between problem-solving skills and programming skills for student success [4, 5, 9] as two different sets of cognitive skills, and suggest to teach problem solving before teaching actual programming. In our curriculum we use that approach, by requiring students to take a problem-solving course before they take their first programming course (see Methods/Participants). Still, to be able to write viable computer programs, students need both problem-solving skills and programming skills. In that context, it is possible that the "algorithmic" factor identified by factor analysis may relate to problem-solving skills. On the other hand, since cluster analysis indicates that the "structural" factor is predictive of students' ability to write viable computer programs, it is also possible that the "structural" factor may relate to the programming skills proposed by others [4, 5, 9]. Beyond the identification of both sets of conceptual categories, our results provide further insight on how different concept categories relate to the students' ability to write viable computer programs.

Our students performed better in conceptual categories related to the "algorithmic" factor than in conceptual categories related to the "structural" factor (Figure 4). This may have been a consequence of the fact that before taking the Programming Fundamentals course (the subject of this study), our students are required to take a problem solving course that emphasizes algorithms to solve problems independently of a programming language. This is consistent with what others have shown that shifting from teaching programming to teaching problem-solving has been shown to increase success in first-year programming courses [4, 5, 9].

However, despite the benefits of an approach teaching problem-solving skills first, the transition from pre-programming problem-solving courses to courses in which students should master a full-fledge programming language remains a challenge [14, 18]. This is reflected in the number of students (41%) who did not have an acceptable performance in both concepts and skills (Figure 1). Even though those students had passed a previous problem solving course, they find the transition to a learning environment that uses a full-fledge programming language like Java difficult. Our results show that a good understanding of concepts related to the "structural" factor may determine whether students would be able to write viable computer programs or not. So, additional emphasis should be placed in concepts related to program structure and organization to facilitate the students' transition from concepts to practical skills.

According to Mayer [13], in addition to the cognitive and metacognitive aspects of problem solving, other aspects like motivation and engagement are also important determinants of student success in problem solving. We believe that student motivation and engagement is an important factor that contributes to the effectiveness of incorporating programming narratives in pre-programming problem solving courses [10, 11]. Therefore, it is likely that pedagogical approaches that motivate and engage students will also facilitate their transition from concepts to practical skills in programming courses, with the concomitant effect on student success.

In interpreting our results, it is important to consider our student population, consisting mostly of underrepresented minorities (see Methods/Participants). Further studies in other institutions will need to be carried out to determine if our results apply to a different context.

## 5. Conclusions

We can conclude: 1) There are two barriers for student success in computer programming courses: a good understanding of programming concepts and the ability to apply those concepts to write viable computer programs; 2) Factor analysis shows that student understanding of computer programming concepts falls in two metaconceptual groups: an "algorithmic" and a "structural" factor; 3) Students have a better understanding of concepts that relate to the "algorithmic" factor than of concepts that relate to the "structural" factor; 4) Student performance in the "structural" conceptual component is predictive of the student's ability to solve practical computer programming problems; 5) Strong emphasis in the structural components of computer programming (assignment, methods, arrays) is necessary for a successful transition from concepts to skills in computer programming courses.

## 6. References


[1] Brooks, R.E. 1997. Towards a theory of the cognitive processes in computer programming. *International Journal of Man-Machine Studies* 9, 737–751. doi:10.1016/S0020-7373(77)80039-4

[2] Cattell, R.B. 1966. The scree test for the number of factors. *Multivariate Behavioral Research* 1, 245-276.

[3] Davies, S.P. 1993. Models and theories of programming strategy. *International Journal of Man-Machine Studies* 39(2), 237–267. doi:10.1006/imms.1993.1061

[4] Deek, F. P., Kimmel, H., and McHugh, J. A. 1998. Pedagogical changes in the delivery of the first-course in computer science: Problem solving, then programming. *Journal of Engineering Education* 87(3), 313–320.

[5] Fincher, S. 1999. What are we doing when we teach programming? In *Proceedings of IEEE Frontiers in Education Conference,* (pp. 12A4/1-12A4/5). IEEE Press.

[6] Guttman, L. 1954. Some necessary conditions for common factor analysis. *Psychometrika* 19, 149-161.

[7] Horn, J.L. 1965. A rationale and test for the number of factors in factor analysis. *Psychometrika* 30, 179-185.

[8] Johnson-Laird, P. N. 1983. *Mental models*. Cambridge, UK: Cambridge University Press.

[9] Kay, J., Barg, M., Fekete, A., Greening, T., Hollands, O., Kingston, J., & Crawford, K. 2000. Problem-based learning for foundation computer science courses. *Computer Science Education* 10(2), 109–128. doi:10.1076/0899-3408(200008)10:2;1-C;FT109

[10] Lansiquot, R. D., and Cabo, C. 2010. The narrative of computing. In *Proceedings of World Conference on Educational Multimedia, Hypermedia and Telecommunications* (Toronto, Canada, June 28-July 01, 2010). ED-MEDIA 2010. AACE, Chesapeake, VA, 3655-3660.

[11] Lansiquot, R. D., and Cabo, C. 2011. Alice's Adventures in Programming Narratives. In C. Wankel and R. Hinrichs (Eds.), *Cutting-edge Technologies in Higher Education, Vol. 4: Transforming Virtual Learning*. Emerald, Bingley, UK, 311-331. DOI: http://dx.doi.org/10.1108/S2044-9968(2011)0000004016.

[12] Lorenzo-Seva, U. and Ferrando, P.J. 2006. FACTOR: A computer program to fit the exploratory factor analysis model. *Behavior Research Methods* 38, 88-91.

[13] Mayer, R.E. 1998. Cognitive, metacognitive, and motivational aspects of problem solving. *Instructional Science* 26, 49-63.

[14] Rist, R. S. 1995. Program structure and design. *Cognitive Science* 19, 507–562. doi:10.1207/s15516709cog1904_3

[15] Robins, A., Rountree, J., and Rountree, N. 2003. Learning and teaching programming. *Computer Science Education* 13, 2, 137-172.

[16] Spohrer, J. C., Soloway, E., and Pope, E. 1989. A goal/plan analysis of buggy Pascal programs. In Soloway, E., & Spohrer, J. C. (Eds.), *Studying the Novice Programmer* (pp. 355–399). Hillsdale, NJ: Lawrence Erlbaum. doi:10.1207/s15327051hci0102_4

[17] Soloway, E., and Spohrer, J. C. (Eds.). 1989. *Studying the novice programmer*. Hillsdale, NJ: Lawrence Erlbaum.

[18] Winslow, L. E. 1996. Programming pedagogy–A psychological overview. *SIGCSE Bulletin* 28(3), 17–22. doi:10.1145/234867.234872