Association for Information Systems

AIS Electronic Library (AISeL)

Proceedings of the 2018 AIS SIGED International Conference on Information Systems Education and Research

SIGED: IAIM Conference

2018

NOVICE PROGRAMMING STRATEGIES

Marjahan Begum
Department of Management, Society and Communication Copenhagen Business School, mbe.msc@cbs.dk

Jacob Nørbjerg
Department of Digitalization Copenhagen Business School, jno.digi@cbs.dk

Torkil Clemmensen
Department of Digitalization Copenhagen Business School, tc.digi@cbs.dk

NOVICE PROGRAMMING STRATEGIES

Marjahan Begum
Department of Management, Society and Communication
Copenhagen Business School
mbe.msc@cbs.dk

Jacob Nørbjerg
Department of Digitalization
Copenhagen Business School
jno.digi@cbs.dk

Torkil Clemmensen
Department of Digitalization
Copenhagen Business School
tc.digi@cbs.dk

ABSTRACT:

This paper identifies novice programmer activities and their implications for the programming outcome. We investigate strategies, cognitive processes and behavior within interacting phases of programming: 1) understanding and design, 2) coding, and 3) debugging and testing. We envision that stronger novice programmers behave differently from weaker novice programmers during the programming process. We develop a questionnaire-based tool, the programming strategy questionnaire (PSQ), which we use to identify the activities novices employ during their development of a program, and we link the strategies to learning outcomes. Finally, we discuss how educators can use our findings to improve the education of novice programmers.

Keywords: novice programmers, programming strategies, programming activities, debugging strategies, program understanding, programming pedagogy

I. INTRODUCTION

The digitalization of work and life highlights the need for educating qualified computer programmers, and programming skills are becoming a core part of both basic school and university level Information Systems curricula. However, despite decades of efforts, students in introductory courses still struggle to learn how to program (Malik, 2018). In this paper we present a questionnaire – the programming strategy questionnaire (PSQ) – to identify cognitive strategies and processes of novice programmers during analysis of the problem domain, design, coding, and debugging programs. We define novice programmers as students who have little or no experience of programming prior to their introductory programming course.

Learning and teaching programming are important not only in CS but also in IS. Even though not all information systems (IS) graduates may become software developers during their careers, they will all play vital roles in managing technologies and understanding how technologies are built (Ngo-ye & Park, 2014), and programming skills are taught both at undergraduate and graduate levels IS programs (vom Brocke, Tan, Topi, & Weinmann, 2017).

The motivation for the study is the high dropout rate among students in both CS and IS, and generally difficulties with learning programming (McCracken et al., 2001; Petersen, Craig, Campbell, & Tafliovich, 2016). Recent studies estimate the pass rate in introductory programming classes around 67% (Watson, 2016). A large number of factors have been identified to this, e.g. lack of motivation and previous similar experience, no commitment to the discipline, and lack of knowledge on programming schema (McCracken et al., 2001; Petersen et al., 2016). Similarly, a large number of approaches to teaching introductory programming have been identified, including: collaborative programming; content change where parts of the teaching material was changed or updated; contextualization, where course content and

activities were aligned towards a specific context; gamification, where a game-themed component was introduced to the course; and grading schema changes (Vihavainen, Airaksinen, & Watson, 2014). The present research, while acknowledging the important factors and innovative approaches identified by others, this research works on the classical premise that if we better understand the interactions between programmer and the programming process, we, as educators, can improve our teaching. More specifically, if we understand the programming behaviours that result in a better outcome; i.e. the strategies adopted by strong novice programmers; we may be able to explicitly cultivate this behaviour. In their early study of programmers, Soloway and Spohrer's stated that the “more we know about what students know, the better we can teach them” (Soloway & Spohrer, 1986). Our approach is that the “more we know about how they program, the better we can teach them" and thus we ask this question: How do novice programmer activities, cognitive strategies and processes through programming process relate to the programming result?

Programming strategies are cognitive processes that result in the programmers’ code being written from scratch, edited or deleted. Thus they are about “how” the knowledge of programming is utilised to solve a programming problems (Davies, 1993; Soloway & Spohrer, 1986). The purpose of identifying the cognitive strategies is that they have influence on the program quality. In an educational context quality is measured by grades in programming courses. Our expectation is that stronger novice programmers consistently use more of certain strategies compared to weak novice programmers. This paper 1) identifies strategies novices use and behaviour they depict; 2) develops a questionnaire for novices to complete after a programming task; and 3) links novice programming strategies to learning outcomes.

II. BACKGROUND - NOVICE PROGRAMMER RESEARCH

We use the terms ‘students’ and 'novice programmers’ interchangeably in our review of the literature, depending on the terms used by the authors.

The difficulties with learning to program are well-documented (Alqadi & Maletic, 2017) and related to a small set of very difficult programming concepts: pointers, references, abstract data types, and error handling (Piteira & Costa, 2013).

Learning programming requires complex problem solving skills that are not used in other areas of life/work. Learning programming is also multi-dimensional because it requires 1) an understanding of the syntax and semantics of a new (programming) language; 2) learning and understanding the logic of fundamental programming concepts (condition, loops, methods etc.); and 3) applying these skills to solving complex problems. All of this takes time to absorb and learn and novice programmers find it very difficult to understand the idea of program structure and designing a program (Piteira & Costa, 2013).

Characteristics of Novice Programmers

A body of work investigates the relationship between the characteristics of novice programmers and their programming results. Novice programmers have many misconceptions and misunderstandings about programming constructs and their meaning, leading to a number of common errors. Among the most frequent mistakes made by novice programmers are unbalanced parentheses, invoking a method with wrong argument(s), and control flow reaching the end of a non-void method without returning (Brown & Altadmri, 2017; Jadud & Dorn, 2015).

Students' problem solving abilities significantly correlate with their performance on programming assignments (Lishinski, Yadav, Enbody, & Good, 2016), and there is a positive relationships between novice programmers’ self-efficacy and success in their CS course (Bhardwaj, 2017).

Novice programmers can develop general problem-solving skills through training in specific heuristic strategies. VanLengen & Maddux (1990) compared the programming behavior and

outcome of two groups of IS students. One group was taught heuristic strategies for program development while the other was taught functions and commands only. However, a post-test measure of problem solving found no significant difference between the two groups. This finding may be related to poor instructions given to students and lack of sensitivity of the test instruments (VanLengen & Maddux, 1990). In contrast, a more recent study in IS showed strong correlation between students' logical reasoning, numerical reasoning and verbal logic and performance in a programming module (Barlow-Jones & van der Westhuizen, 2017). No conclusive evidence, however, firmly links specific novice programmer traits to programming performance.

Novice Programmer Behavior

While extant research has studied the relationship between programming performance and novice programmers' difficulties, knowledge, and personal characteristics, we know less about the actual behavior of programmers through the programming process; from understanding the problem specification to having constructed a computer program (Vihavainen, Helminen, & Ihantola, 2014).

Meta-cognition and self-regulation is relevant in the programming process. Metacognition is an individual's knowledge of their own cognitive processes and their ability to control these processes by organizing, monitoring and modifying them as a function of learning (Davidson, Deuser, & Sternberg, 1994). Bergin, Reilly, & Trayno (2005) argues that self-regulated learning incorporates cognitive and meta-cognitive strategies in addition to resource management strategies and motivational aspects. They divided the cognitive strategies into rehearsal, elaboration, and organization strategies and found that while there were positive correlations between meta-cognitive strategies and programming, there were no correlations between cognitive strategies and programming performance (Bergin, Reilly, & Traynor, 2005). Their classification does not, however, reflect the cognitive strategies identified by other studies of novice programmers such as reading and understanding the problem specification, designing code, and evaluating code (Booth, 1992; Marton, 1981).

Investigation into meta-cognition and program comprehension has shown that the use of meta-cognition influences how well programmers understand a program (Shaft & Vessey, 1998). This resonates with more recent research into, for example 'pattern matching' where students seek similar examples or recognize similar code from somewhere else (Fitzgerald et al., 2008, 2010; McCauley et al., 2008).

Bishop-Clark (1995) took a holistic approach to studying novice programmers, and investigated the effect of cognitive styles and personality traits on different phases of programming. She concluded that different cognitive styles influence in different programming phases. Distinguishing factors between cognitive styles and strategies are, first, that of the ingrained characteristics of a programmer and, second, the ways in which the programming process is completed (Bishop-Clark, 1995).

Debugging

Novice programmers spend a substantial part of their programming efforts in debugging and the activity receives much attention (Ducasse, M., & Emde, 1988; Fitzgerald et al., 2008, 2010). Debugging is, however, difficult for novice programmers because they only have limited debugging knowledge. They also often have difficulty understanding the code they have written themselves when they discover that the actual program does not behave according to the problem specification (Ducasse, M., & Emde, 1988).

Novice programmers have very limited knowledge of the programming language and as a result, the bug localization and bug repair phases take longer. They find deciphering compiler error messages very time-consuming and difficult. Part of this difficulty can be attributed to the

lack of programming knowledge and partly to a limited understanding of compiler errors (Ducasse, M., & Emde, 1988).

Novice programmers have misconceptions about programming logic, such as failing to use an intermediate variable when swapping values between variables, and they often introduce logical bugs into their program (Du Boulay, O'shea, & Monk, 1999).

Summary

While extant research about novice programmers provides insights both into programmer characteristics and the challenges novice programmers face during the different stages of programming, there is a gap in our knowledge about how novice programmer activities, cognitive strategies and processes through the whole programming process relate to the programming result. In this research, we refer to "programming strategies" to encompass all of these activities i.e.: cognitive processes, general strategies and programmers' behavior.

III. COGNITIVE PROGRAMMING STRATEGIES

Different cognitive strategies are associated with a programmer's individual activity and carried out within different programming sub-processes: (1) understanding the problem; (2) designing and coding the program; (3) debugging and testing. The sub-processes are interacting and iterative. The aim of this research is to identify the cognitive and meta-cognitive strategies used by novice programmer throughout the programming sub-processes and the impact of these strategies on the quality of the program, see Figure 1.

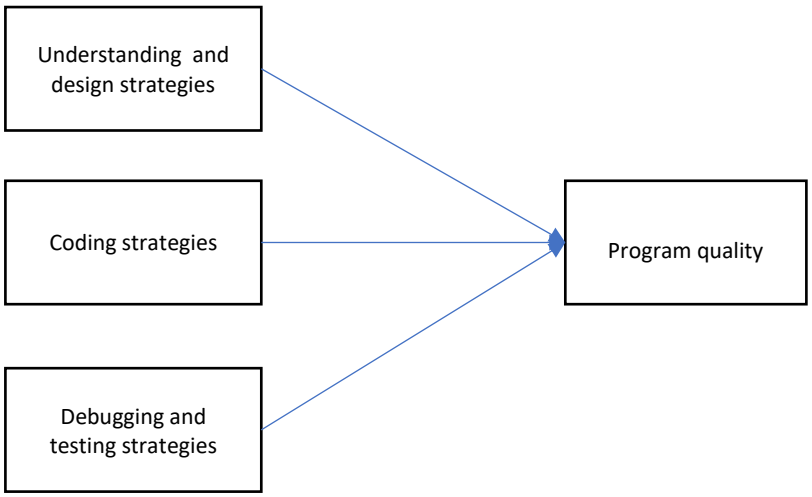

Figure 1: Research model

In the following, we will develop the concepts and constructs used to study the cognitive processes of novice programmers. Existing frameworks distinguish understanding and coding from debugging. We will follow this distinction and discuss the first two sub-processes, understanding, design, and coding together, while the third process, debugging and testing, is discussed in a separate sub-section.

Understanding, design, and coding

Previous research into programming activities and strategies include, e.g., (Barlow-Jones & van der Westhuizen, 2017; Booth, 1992; Ducasse, M., & Emde, 1988; Katz & Anderson, 1987; Kessler, C. M., & Anderson, 1986; VanLengen & Maddux, 1990). Booth (1992) identified four

distinct approaches to solving a programming problem (Booth, 1992; Marton, 1981). See Table 1.

Table 1. The Four Approaches to Programming (Booth 1992)

Programming approach	Description
Expedient	Produce a complete program from the outset by making use of an existing program or by adopting some known program
Constructional	Recognize details of the problem in terms of features of the programming language – construct, functions and keywords – which can be used to build a program
Operational	Write a program based on an interpretation of the problem within the domain of programming; the problem is considered from the point of view of what operations the program has to perform
Structural	Write a program based on an interpretation of the problem within its own domain; the structure of the problem is considered and on that basis a program is devised

Novice programmers who used structural and operational approaches to writing programs adopted deep approaches to learning, while those who used constructional and expedient approaches adopted surface approaches. The evidence also showed that there was a correlation between the programming approaches and exam results. In particular, students who adopted structural and operational approaches performed better than those who adopted constructional and expedient approaches (Booth, 1992).

Debugging and testing

Figure 2 depicts debugging as an iterative process with four activities (Kessler, C. M., & Anderson, 1986).

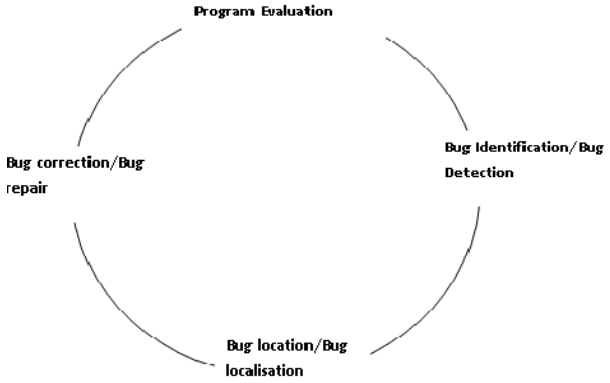

Figure 2: The Behavior of Novice Programmers During the Debugging Phase (Kessler, and Anderson, 1986).

For a typical novice programmer, the evaluation of the program initiates the debugging process, i.e. reading the code, compiling source code, or testing. The program evaluation process can also begin when the programmer has finished a piece of code or a sub-routine.

Once the programmer is satisfied that there are no syntactical errors, the program is executed and evaluated with test data, and the results are compared to the correct output (as per the problem specification). If the program does not deliver the expected output, the debugging process starts from program comprehension.

Successful debugging depends on a wide range of knowledge (Table 2) (Ducasse, M., & Emde, 1988). Debugging methods are strategies novice programmers utilize as a result either of having the debugging knowledge or of building this knowledge.

Novice programmers find debugging difficult because they only have limited debugging knowledge, as compared to the extant range of necessary knowledge about debugging. (Ducasse, M., & Emde, 1988).

Extant debugging research asks programmers to debug completed programs, while in our research, novice programmers gain knowledge of the domain and the intended program behavior by studying the problem specification and developing the program. However, novice programmers have very limited knowledge of the programming language and they have difficulties understanding even their own code. As a result, the bug localization and bug repair phases take longer, e.g., (Ducasse, M., & Emde, 1988). Thus, our study of novice programmers' debugging activities will focus on how novice programmers build knowledge about the program's behavior and how to correct it.

Table 2. Debugging Knowledge (Ducasse, M., & Emde, 1988).

Knowledge type
Knowledge of the intended program
Program I/O
Behavior
Implementation
Knowledge of the actual program
Program I/O
Behavior
Implementation
Understanding of the programming language
General programming expertise
Knowledge of the application domain
Knowledge of bugs
Knowledge of debugging methods

IV. RESEARCH METHOD

Table 3 outlines the cognitive strategies investigated in the study. It is based on previous research as well as our own observations of novice programmers. The cognitive strategies include tasks associated with the direct interaction between the programming environment and the programmer. The strategies are at both micro and macro levels of interactions. The Programming Strategy Questionnaire (PSQ) also includes behavioral elements (e.g. taking break) and meta cognitive strategies (e.g., assessing clarity level of the problem).

A number of statements were identified for each of the constructs in table 3. Three statements were, for example, identified for the construct: Focusing on understanding the problem:

"Before coding and designing, I read the problem specification more than once", "*It was important for me to understand how to solve the problem without thinking about the Java syntax*" and "*I made sure I understood the problem before starting any coding*". 22 statements were identified for the Understanding and design phase, 27 for Coding, and 33 for Testing.

In the table, '+' and '-' indicate that using the strategy was found to have a positive or negative effect on results respectively. This is explained in detail in the results section.

Table 3: Categories of Activities within Iterative Phases of Programming.

Understanding and Design strategies	**Coding strategies**	**Debugging and Testing strategies**
Assessing level of difficulty	Compilation and execution driven coding (-)	Checked computational equivalence of intended and actual program
Assessing clarity level	Non evaluative coding (-)	Filtering – tracing actual program for understanding data behavior and implementation
Evaluating problem and solutions options (+)	Clarifying the problem specification (+)	Filtering - program slicing actual program to narrow source of error and for understanding program implementation
Focusing on understanding the problem	Considering code reuse	Desk-checking the actual program for understanding data, behavior and implementation (+)
Considering lower level requirement	Amending code (+)	Mentally filtering the actual program understanding program data flow and program behavior
Focusing on operational aspect of the program	Operational coding	Commenting to increase readability and help design/coding
Simplifying the problem	Evaluative coding	Difficulties with debugging (-)
Identifying key information in the specification (+)	Confused and frustrated (-)	Evaluating program correctness using inventive and extreme test data (+)
Using pen and paper (-)	Incautious coding (-)	Evaluating program correctness using given test data (+)
	Having lower level difficulties (-)	Modular testing (+)
	Considering lower level requirement (+)	Evaluating program correctness using trial and error strategy
	Having higher level difficulties (-)	Recognizing stereotypical errors
	Short-cut coding (+)	Use pen and paper to identify problems(-)
		Taking a break

The Scale of Items

Each question is measured on a Likert scale with six items: "Definitely applies to me", "Probably applies to me", "Not sure if this applies or not", "Probably does not apply to me",

"Definitely does not apply to me" and "Don't know what this means". The last phrase was used to identify ambiguous items in the questionnaire during pilot testing. Such items were re-evaluated or rephrased before subsequent administration of the questionnaire. If the respondents still rated an item with "Don't know what this means", the respective item was not included in further analysis.

The questionnaire was developed in several iterations and tested with different groups (CS and non-CS) students before administering to the participants in this research.

Reliability

We tested the reliability of the PSQ with *test-retest reliability* and *internal consistency reliability*. Test-retest was carried out by letting students complete the questionnaire after completing the programming problem, and again after a period of time. Assuming the collective items in each of the above groups of phases measure student behaviour in term of cognitive strategies, the total values from the first administration of the questionnaire must be correlated with the total values from the second administration. The correlation coefficient was calculated as 0.83 with a significance level at 0.006 for *"Understanding and Design"*, while for *"Coding"* it was 0.93 with a significance level at 0.000 and finally for "Debugging and Testing" it was 0.94 with a significance level at 0.000.

Internal consistency reliability is concerned with the homogeneity of the items within the scale (De Vellis, 2003). Items that can be logically grouped together will intercorrelate with each other significantly and these logical groups are called theoretical groups. In this research theoretical groups are each of the strategies as illustrated in table 3. For example the cognitive strategies within these strategies : "evaluating problem and solution option", "using pen and paper", "compilation and execution driven coding" and "non-evaluative coding" had high internal consistency reliably (i.e., greater than 0.5). Interestingly for the cognitive strategy "desk-checking the actual program for understanding data, behaviour and implementation" is had a very high value of 0.90.

Validity

The PSQ was tested for *content validity* and *criterion validity*. Content validity involved asking experts to assess if the items in the questionnaire measure the construct. For example if the item *"I thought the problem was too difficult for me"* measures the construct "*Level of difficulty*". The results are shown in Table 4.

Table 4. Content Validity.

Phases in programming	**% of items expert agreed**
Understanding and Design phase	68%
Coding phase	85%
Debugging and Testing phase	90%

Criterion validity was established by making logical assumptions about the theoretical groups identified. One such assumption is that *"Using pen and paper"* in the "Understanding and Design" phases would correlate to *"Using pen and paper"* in the "Debugging and Testing" phases. This is because it is assumed that if someone is using pen and paper they would consistently use these during all the phases. In this case the correlation coefficient was calculated as 0.72.

Data Collection

The PSQ was tested with CS undergraduate students. The questionnaire was distributed twice after the students had solved a programming problem. The first time after 5 weeks of study and the second after 10 weeks. The aim was to identify to what extent students can relate to using the strategies identified in the questionnaire as well as to obtain preliminary insights into the relationship between cognitive strategies and outcomes among novice programmers.

There were 129 students[1] in the course of which 99 (77%) completed the first questionnaire and 88 (68%) the second. Completing the questionnaire took approximately 30 minutes.

Students were to reflect back on their programming process and rate each item in the questionnaire. The marks from the programming exercises contributed to their first semester result. The semester result and individual marks for the problems were also collected for this study. No marks were given for completing the questionnaire.

V. RESULTS

The combined effect of the explanatory variables (novice programmer behavior and strategies) on the dependent variable of learning outcome (student performance) was investigated with regression analysis. Regression analysis takes account of possible correlations among the effects of the many explanatory variables in the study.

The results show how the students' strategies influence the quality of the program, i.e., the grades. We admit that cognitive processes alone cannot account for students' performance, but we are, on the other hand, unable to include all possible variables (whether measurable or not, observable or not) in the study. However, we can assume some of the variation in performance to be explained by the explanatory variables.

Several regression models were generated in order to identify the variables with the highest significance. Space does not allow a complete account of the results of this process. We limit the presentation to the most important relationships identified. The importance of a relationship is indicated by the frequency in which it appears with high significance in the regression models.

The results do not to offer an all-encompassing predictive model but indicate what strategies contribute positively and negatively to novice programmer performance.

Understanding and Design

Two explanatory variables stand out in the Understanding and Design phase (Figure 3): “Visualizing the solution of the problem” had a significant positive effect on the outcome. This is because the 21 regression models generated for this variable had strong effect on the results for 17 of the models, hence 83%. On the other hand, “Drawing picture or model of the problem” surprisingly had a 57% negative effect.

[1] Only 10% of the students had some, though often limited, experience of programming using the Java programming language.

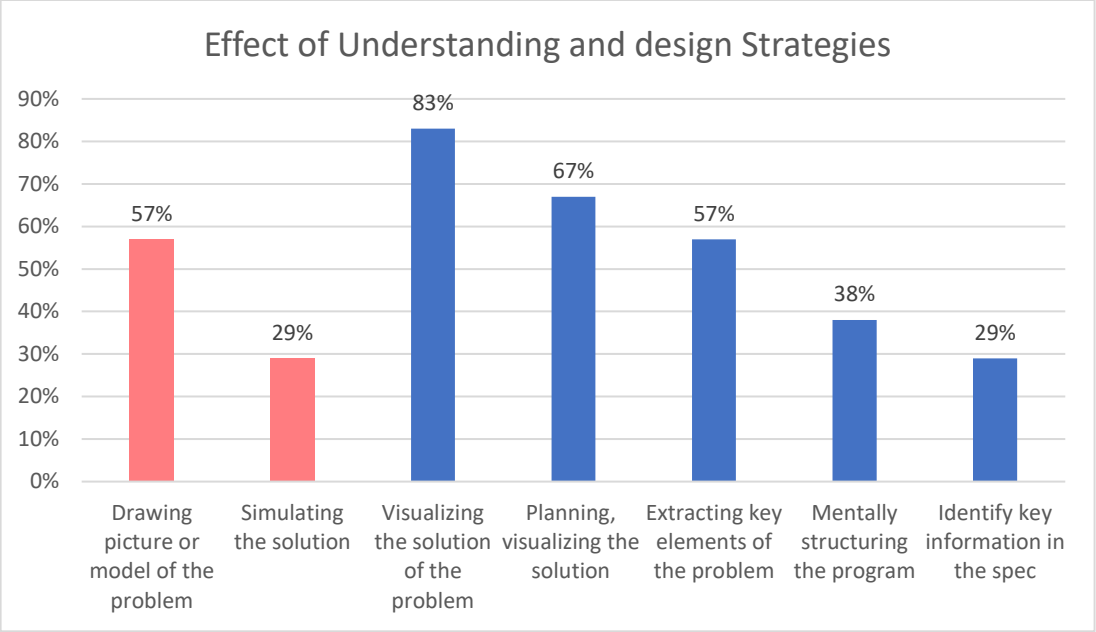

Figure 3: % of Regression Models Showing Significant Relationship Between Strategies and Results During Understanding and Design (Red: negative effect; Blue: positive effect)

This result suggests that only struggling students chose to use pen and paper and if a strong novice programmer happened to use pen and paper it did not affect their results. Use of pen and paper does not suggest effectiveness of the design and it may be important to investigate further what is drawn.

In terms of broader strategies, it appears that stronger novice programmers tend to use more of "evaluating problem and solutions options" and "identifying key information in the specification" and weaker novice programmers tend to use more of "using pen and paper" (see Table 3).

Coding

Figure 4 illustrates findings for the Coding process. Two specific behavior "*Having difficulties with program integration (output driven)*" and "*Having doubt about the progress*" has a significant negative influence on outcome.

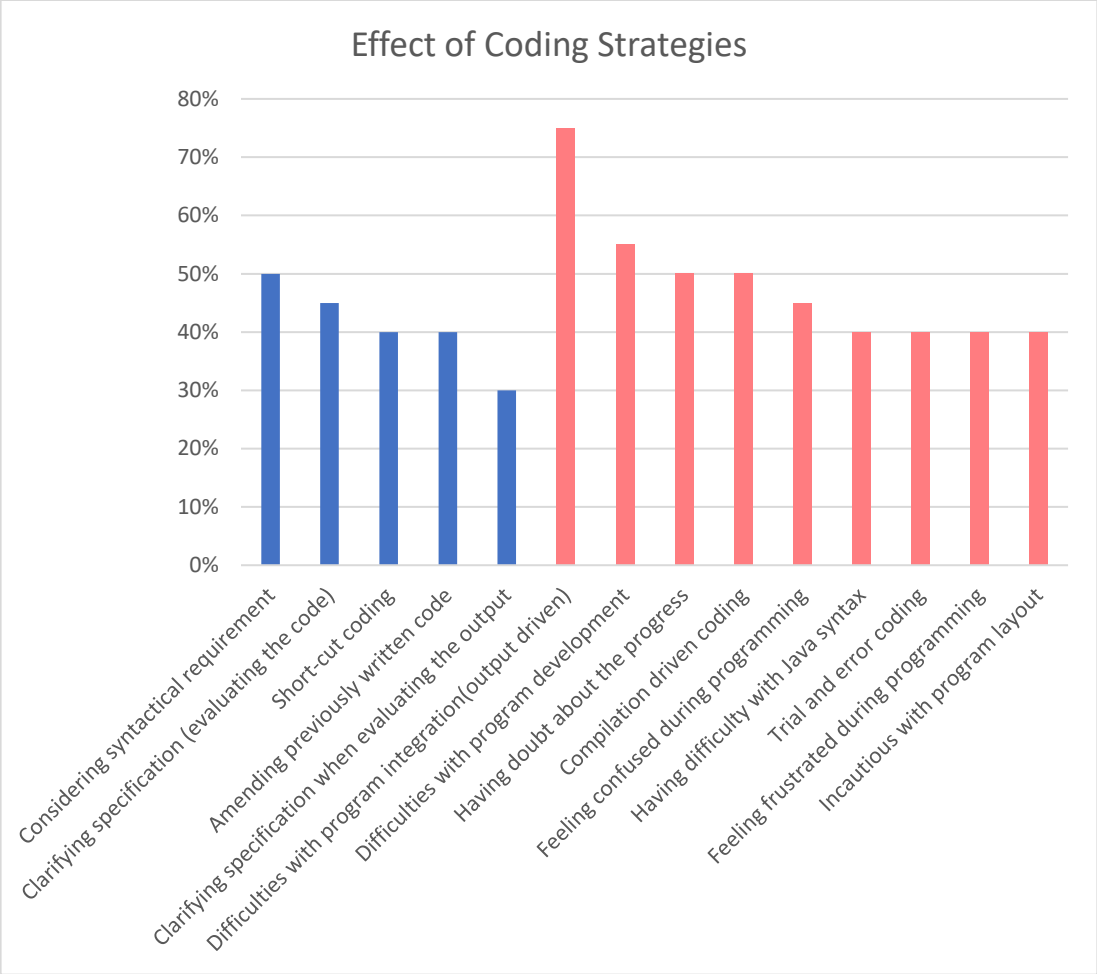

Figure 4: % of Regression Models Showing Significant Relationship Between Strategies and Results During coding **(**Red: negative effect; Blue: positive effect)

This suggests that students who got lower marks had difficulties integrating different parts of the program and inevitably were not sure about their progress. The result suggests that this aspect may have affected their performance. In terms of broader strategies, weaker programmers uses more of "non evaluative coding, were "confused and frustrated", did more "incautious coding", was "having lower level difficulties" and "having higher level difficulties" during coding (see Table 3). On the other hand stronger novice programmers use more of "short-cut coding", "amending code", "considering lower level requirement" and "clarifying the problem specification" (see Table 3).

Debugging and Testing

Figure 5 shows results during debugging and testing across 21 sub-models. Explanatory variables that have a positive relation with marks are *"Evaluating program correctness - using given test data"* and *"Evaluating program correctness using modular testing"*. In general with respect to broader strategies, that stronger novice programmer employs more of are "desk-checking the actual program for understanding data, behavior and implementation, "evaluating program correctness using inventive and extreme test data", "evaluating program correctness using given test data" and "modular testing" (see Table 3).

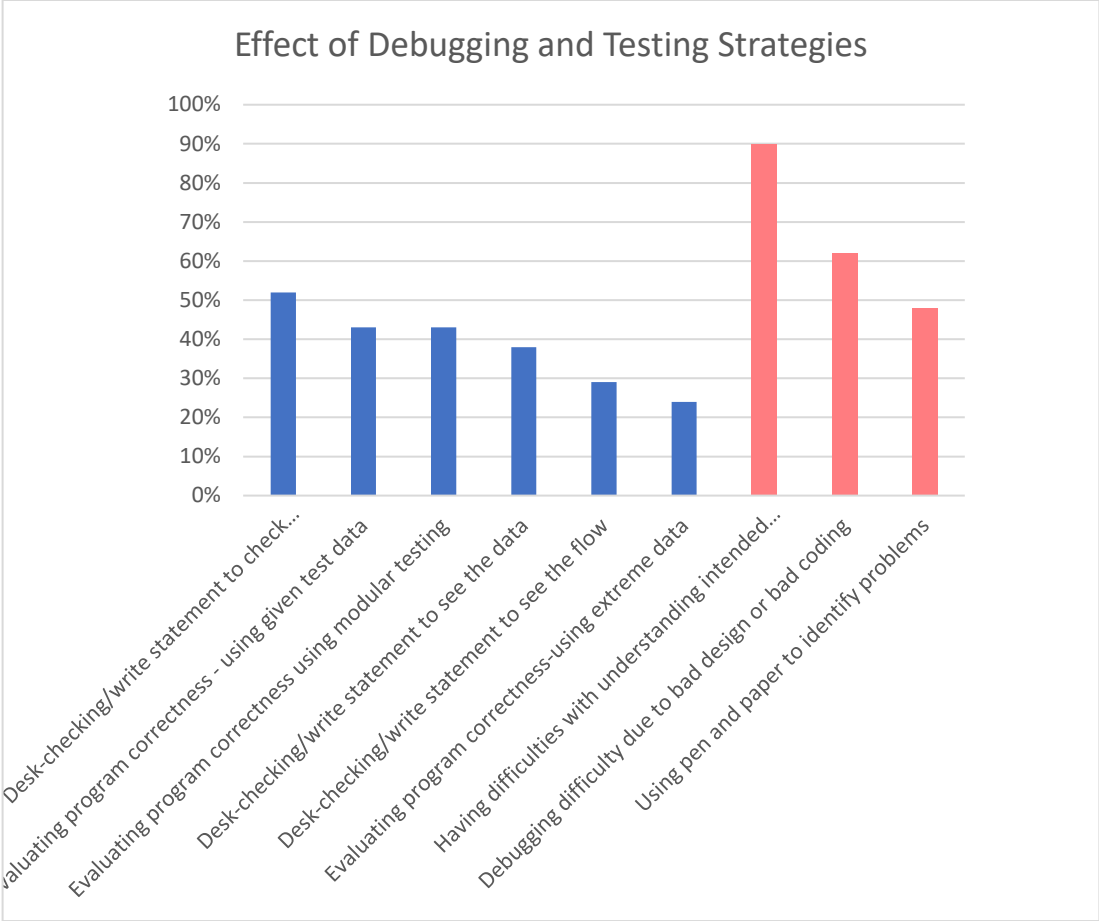

Figure 5: % of Regression Models Showing Significant Relationship Between Strategies and Results During Debugging and Testing (Red: negative effect; Blue: positive effect)

Conversely, “having difficulties with understanding intended program implementation”, “having doubt about the progress of the program development”, “having difficulties with debugging”, “debugging difficulty due to bad design or bad coding” and “using pen & paper to identify problems” were all negatively related to results.

The significance of the results is that a stronger novice programmer would use on average more strategies that contribute to having a high positive coefficient and less of those strategies that has high negative coefficient. These findings are important as they assist in generalizing about novice programmers’ behavior for the population of this study.

VI DISCUSSION

This research was motivated by an interest in improving the introduction to programming education by creating learning environments that help novice programmers develop programming strategies, which contribute to better programming results. The objective of the programming strategy questionnaire (PSQ) used is to identify strong novice programming strategies that link to positive learning outcomes. Our expectation was that strong novice programmers consistently use more of certain strategies than their weaker peers. We 1) identified strategies novice programmers use; 2) developed a questionnaire for novices to complete after a programming task; and 3) linked novice programming strategies to learning outcomes. Below we discuss each of these in turn.

Novice programming strategies

During the understanding and design phase strategies associated with “evaluating problem and solutions options” and “identifying key information” such as planning and visualizing the solution and extracting key elements of the problem help novice programmers produce better quality code, i.e., these strategies had a positive effect on most aspects of the learning outcome. On the other hand, strategies associated with “using pen and paper” such as drawing picture or model of the problem had a negative effect. This sound paradoxical, and we discuss it below in the section on linking strategies to learning outcomes.

In the coding phase our findings indicate that if novice programmers may need extra support if they have difficulties integrating the program. Novice programmers, who got lower marks, had difficulties integrating different parts of the program and inevitably were not sure about their progress. In contrast, it appears that novice programmers who reuse code, check the specification while coding, and have knowledge of a simpler way of coding, got better marks. This finding is in line with (Bonar & Soloway, 1985).

In the debugging and testing phase, desk checking with write statement to understand flow of data and flow of control leads to positive marks. Not understanding the intended program implementation or logic leads to negative marks. The latter result is in line with earlier findings about code comprehension and programming logic misconceptions (Du Boulay et al., 1999).

The properties of the programming strategy questionnaire (PSQ)

Our results imply that the PSQ has strong validity and reliability with respect to the coding, debugging, and testing activities, but it is comparatively weaker regarding the understanding and design phase. It is not surprising that coding, and debugging and testing have higher reliability and validity, as these phases are easier to observe than understanding and design. There is also a large body of prior research about coding, debugging and testing from which one can build adequate constructs (Booth, 1992; Ducasse & Emde, 1988; Fitzgerald et al., 2008, 2010; Kessler, C. M., & Anderson, 1986; McCauley et al., 2008).

Linking optimal programming strategies to positive learning outcomes

We are still a long way from being able to firmly distinguish strong and weak novice programmer strategies. Our main assumption is that that a strong novice programmer uses on average more strategies with a positive effect on marks and less with a negative. Our initial results suggest that some strategies are more effective for more advanced problems. As the problem gets more difficult, planning becomes more significant. Also, specification of harder problems requires more scrutiny, and identifying the syntax becomes an important process.

It is important to point out that using pen and paper appears to distinguish weak novice programmers. The result suggests that struggling students chose to use pen and paper while the use of pen and paper by stronger students has no effect. Our results show that “Visualizing the solution of the problem” had a significantly positive effect on the outcome, while “Drawing picture or model of the problem” had a surprising negative effect. A possible explanation may lie in the early studies of programming, which connected mental imagery of programmers with the visual sketches used in other design disciplines (Petre & Blackwell, 1999) – something our initial results cannot support. Perhaps weak novice programmers use pen and paper to show they do 'something' or that they do 'as told'. Use of pen and paper does not suggest effectiveness of the design and it may be important to investigate further, what is drawn. Thus, further research aims to refine the PSQ tool and continue to investigate novice programmer behavior in different settings.

VII CONCLUSION

This research presents the development and application of a questionnaire to examine the strategies novice programmers use in different phases of programming. An initial validation and reliability of the PSQ was carried out in a CS1 course. Results indicate that the questionnaire has some validity and reliability. It will need to be further developed and used in future research on programming strategies.

Findings are in line with previous research on programming: Teaching students explicitly how to plan and how to debug using the knowledge of control and data flow may enable the transition from being a weak novice to a strong novice programmer. In addition, once the questionnaire is further developed, it can be used during courses to identify students who require extra support on learning explicit strategies.

VIII. REFERENCES

Alqadi, B. S., & Maletic, J. I. (2017). An Empirical Study of Debugging Patterns Among Novices Programmers. In *Proceedings of the 2017 ACM SIGCSE Technical Symposium on Computer Science Education - SIGCSE '17* (pp. 15–20). New York, New York, USA: ACM Press. https://doi.org/10.1145/3017680.3017761

Barlow-Jones, G., & van der Westhuizen, D. (2017). Problem solving as a predictor of programming performance. In *Communications in Computer and Information Science* (Vol. 730, pp. 209–216). https://doi.org/10.1007/978-3-319-69670-6_14

Bergin, S., Reilly, R., & Traynor, D. (2005). Examining the role of self-regulated learning on introductory programming performance. *First International Workshop on Computing Education Research*, 81–86. https://doi.org/10.1145/1089786.1089794

Bhardwaj, J. (2017). In search of self-efficacy: development of a new instrument for first year Computer Science students. *Computer Science Education*, *27*(2), 79–99. https://doi.org/10.1080/08993408.2017.1355522

Bishop-Clark, C. (1995). Cognitive style, personality, and computer programming. *Computers in Human Behavior*, *11*(2), 241–260. https://doi.org/10.1016/0747-5632(94)00034-F

Bonar, J., & Soloway, E. (1985). Preprogramming Knowledge: A Major Source of Misconceptions in Novice Programmers. *Human-Computer Interaction*, *1*(2), 133–161. https://doi.org/10.1207/s15327051hci0102_3

Booth, S. (1992). *Learning to program: A phenomenographic perspective*. University of Gothenburg.

Brown, N. C. C., & Altadmri, A. (2017). Novice Java Programming Mistakes: Large-Scale Data vs. Educator Beliefs. *Trans. Comput. Educ.*, *17*(2), 7:1--7:21. https://doi.org/10.1145/2994154

Davidson, J. E., Deuser, R., & Sternberg, R. J. (1994). The Role Of Metacognition In Problem Solving. In J. Metcalfe & A. P. Shimamura (Eds.), *Metacognition:Knowing About Knowing* (pp. 207–226). The Mit Press.

Davies, S. P. (1993). Models and theories of programming strategy. *International Journal of Man-Machine Studies*, *39*, 237–267.

De Vellis, R. F. (2003). *Scale Development: Theory and Applications* (2nd Editio). Thousand Oaks, CA: Sage Publications.

Du Boulay, B., O'shea, T., & Monk, J. (1999). The black box inside the glass box: presenting computing concepts to novices. *International Journal of Human-Computer Studies*, *51*(2), 265–277. https://doi.org/10.1006/IJHC.1981.0309

Ducasse, M., & Emde, A.-M. (1988). A review of automated debugging systems: Knowledge,strategies and techniques. In I. W. S. & P. B. (Eds.) (Ed.), *Proceedings of the*

10th international conference on software engineering (pp. 162–171). Singapore: IEEE Computer Society Press.

Ducasse, M., & Emde, A.-M. (1988). A review of automated debugging systems: knowledge, strategies and\ntechniques. *Proceedings. [1989] 11th International Conference on Software Engineering*, 162–171. https://doi.org/10.1109/ICSE.1988.93698

Fitzgerald, S., Lewandowski, G., McCauley, R., Murphy, L., Simon, B., Thomas, L., & Zander, C. (2008). Debugging: finding, fixing and flailing, a multi-institutional study of novice debuggers. *Computer Science Education*, *18*(2), 93–116. https://doi.org/10.1080/08993400802114508

Fitzgerald, S., McCauley, R., Hanks, B., Murphy, L., Simon, B., & Zander, C. (2010). Debugging from the student perspective. *IEEE Transactions on Education*, *53*(3), 390–396. https://doi.org/10.1109/TE.2009.2025266

Jadud, M. C., & Dorn, B. (2015). Aggregate Compilation Behavior. *Proceedings of the Eleventh Annual International Conference on International Computing Education Research - ICER '15*, 131–139. https://doi.org/10.1145/2787622.2787718

Katz, I., & Anderson, J. (1987). Debugging: An Analysis of Bug-Location Strategies. *Human-Computer Interaction*, *3*(4), 351–399. https://doi.org/10.1207/s15327051hci0304_2

Kessler, C. M., & Anderson, J. R. (1986). A model of novice debugging in LISP Title. In & S. I. E. Soloway (Ed.), *Empirical Studies of Programmers.* Norwood, NJ: Ablex.

Lishinski, A., Yadav, A., Enbody, R., & Good, J. (2016). The Influence of Problem Solving Abilities on Students' Performance on Different Assessment Tasks in CS1. *Proceedings of the 47th ACM Technical Symposium on Computing Science Education - SIGCSE '16*. https://doi.org/10.1145/2839509.2844596

Malik, S. I. (2018). Improvements in Introductory Programming Course: Action Research Insights and Outcomes. *Systemic Practice and Action Research*. https://doi.org/10.1007/s11213-018-9446-y

Marton, F. (1981). PHENOMENOGRAPHY -DESCRIBING CONCEPTIO WORLDAROUNDUS. *Instructional Science*, *10*, 177–200.

McCauley, R., Fitzgerald, S., Lewandowski, G., Murphy, L., Simon, B., Thomas, L., & Zander, C. (2008). Debugging: a review of the literature from an educational perspective. *Computer Science Education*, *18*(2), 67–92. https://doi.org/10.1080/08993400802114581

McCracken, M., Wilusz, T., Almstrum, V., Diaz, D., Guzdial, M., Hagan, D., … Utting, I. (2001). A multi-national, multi-institutional study of assessment of programming skills of first-year CS students. *ACM SIGCSE Bulletin*, *33*(4), 125. https://doi.org/10.1145/572139.572181

Ngo-ye, T., & Park, S. (2014). Motivating Business Major Students to Learn Computer Programming – A Case Study.

Petersen, A., Craig, M., Campbell, J., & Tafliovich, A. (2016). Revisiting why students drop CS1. *Proceedings of the 16th Koli Calling International Conference on Computing Education Research - Koli Calling '16*, 71–80. https://doi.org/10.1145/2999541.2999552

Petre, M., & Blackwell, A. F. (1999). Mental imagery in program design and visual programming. *Int. J. Human-Computer Studies*, 267. Retrieved from http://www.idealibrary.comon

Piteira, M., & Costa, C. (2013). Learning computer programming. In *Proceedings of the 2013 International Conference on Information Systems and Design of Communication - ISDOC '13* (p. 75). New York, New York, USA: ACM Press. https://doi.org/10.1145/2503859.2503871

Shaft, T. M., & Vessey, I. (1998). The Relevance of Application Domain Knowledge: Characterizing the Computer Program Comprehension Process. *Journal of Management*

Information Systems, *15*(1), 51–78. https://doi.org/10.1080/07421222.1998.11518196

Soloway, E., & Spohrer, J. C. (1986). Novice mistakes: are the folk wisdoms correct? *Communications of the ACM*, *29*(7), 624–632.

VanLengen, C. A., & Maddux, C. D. (1990). Does instruction in computer programming improve problem solving ability? *CIS Educator Forum*.

Vihavainen, A., Airaksinen, J., & Watson, C. (2014). A Systematic Review of Approaches for Teaching Introductory Programming and Their Influence on Success. In *Proceedings of the Tenth Annual Conference on International Computing Education Research* (pp. 19–26). New York, NY, USA: ACM. https://doi.org/10.1145/2632320.2632349

Vihavainen, A., Helminen, J., & Ihantola, P. (2014). How novices tackle their first lines of code in an IDE. In *Proceedings of the 14th Koli Calling International Conference on Computing Education Research - Koli Calling '14*. https://doi.org/10.1145/2674683.2674692

vom Brocke, J., Tan, B., Topi, H., & Weinmann, M. (2017). *The Global Report of the Association for Information Systems on Information Systems Education 2017*.

Watson, C. (2016). Version of attached le : Failure Rates in Introductory Programming Revisited. *Proceedings of the 2014 Conference on Innovation Technology in Computer Science Education (ITiCSE'14)*, *44*(July), 0–6. https://doi.org/10.1145/2591708.2591749