Assessing Programming Behaviors Through Evidence-Centered Design

Roya Hosseini

Intelligent Systems Program

University of Pittsburgh

Advisor: Dr. Peter Brusilovsky

June 7, 2016

Assessing Programming Behaviors Through Evidence-Centered Design

Over the last 10 years, the popularity and use of systems for automated assessment of programming assignments have been growing constantly. From the pioneer Ceilidh (Benford et al., 1993) to the modern Web-CAT (Edwards et al., 2008), these systems offer instructors and students highly valuable functionality: the ability to assess the correctness and other important features of student programs without investing a lot of manual labour. Because of this functionality, these systems have enabled instructors to increase the number of programming assignments in a regular course and offer multiple submission attempts for each assignment (Douce et al., 2005).

However, another important impact of automatic assessment systems is being discussed much more rarely: their ability to increase our understanding of how humans solve programming problems. Course by course, automatic assessment systems accumulate many program submissions. These submissions open a window on students' problem solving process on both the personal and community levels. On the personal level, the analysis of single student program submissions could reveal *programming behavior*—how the student progresses to the correct solution through several incorrect solutions and how his or her knowledge grows from assignment to assignment. On the community level, the analysis of submissions for a single problem could reveal multiple correct and incorrect ways to solve the same problem.

Active work on community-level analysis has already started, propelled to a large extent by the availability of the very large volume of problem solutions collected by several programming massive open online courses (MOOCs) that apply automatic assessment. Two interesting papers demonstrate how the MOOC data could be used to analyze the landscape of students' solutions for individual problems (Huang et al., 2013; Piech et al., 2012), while another shows how these data could be used to build an intelligent scaffolding system (Rivers & Koedinger, 2013).

This paper contributes to the analysis of program submission on the personal level. We use evidence-centered design (ECD) to describe how to assess programming behavior by analyzing students' program construction activities. We describe the main components of the ECD framework and explain how identifying programming behavior can map onto these layers. We then discuss the findings from our first attempt at identifying students' programming behavior, elaborate on the implications of our approach, and close with future directions for this research.

The Components of Evidenced-Centered Design

What We Are Measuring: The Student Model

Focal Construct

The focal construct addressed in this design framework is problem solving in the domain of programming. Understanding and identifying factors that affect students' programming skills have been of interest to computer science educators for decades. Work on identifying traits that indicate a tendency to be a successful programmer dates back to the 1950s (Rowan, 1957). Since then, dozens of factors have been investigated, such as aptitude in mathematics (Byrne & Lyons, 2001; White & Sivitanides, 2003) and science subjects (Byrne & Lyons, 2001), intrinsic motivation and comfort level (Bergin & Reilly, 2005), and learning styles (Thomas, Ratcliffe, Woodbury, & Jarman, 2002). More recent research, however, has suggested that many of the factors might be context specific, thus encouraging researchers to consider data-driven approaches to investigate students' ability to solve programming problems (Watson, Li, & Godwin, 2014). One such approach is the collection and analysis of online protocols, which typically involve gathering information by augmenting the programming environments students use to write, compile, and test their programs (Tabanao, Rodrigo, & Jadud, 2011).

So far, programming trace logs have been used to understand student coding behavior or abilities (Blikstein, 2011; Hosseini, Vihavainen, & Brusilovsky, 2014; Watson et al., 2014), model student's knowledge in a program development context (Yudelson, Hosseini, Vihavainen, & Brusilovsky, 2014), and predict student success (Vihavainen, 2013; Vihavainen, Luukkainen, & Kurhila, 2013) and grade performance (Murphy, Kaiser, Loveland, & Hasan, 2009). The focus of this paper is on the first application, using trace logs to understand student programming behavior.

Understanding programming behaviors using observational data collected from students' problem solving activities is beneficial for both instructors and students. It enables instructors to better address concepts that students find difficult to grasp or are having misconceptions about (Tabanao et al., 2011). More important, according to Tabanao and colleagues, it helps identify and provide targeted help and proper intervention for struggling students who have a tendency to become discouraged and frustrated by their mistakes and thus give up after some trials and failures. Therefore, identifying programming behavior may help mitigate students' frustration and confusion, increase programming comprehension, and contribute to greater student retention in the course. In addition to these benefits, this information could be used to provide proactive recommendations and suggestions for all students, such as how long to expect an assignment to take or what errors to look out for (Murphy et al., 2009).

This paper describes a data-driven programming behavior assessment, introduced in Hosseini et al. (2014) from the perspective of ECD. This approach resulted in a deep conceptual analysis of students' intermediate programming steps and thus captured students' coding behavior better than approaches that consider only the end result of the code or errors made during coding.

Knowledge, Skills, and Abilities (KSAs)

The common confounding factors (i.e., knowledge, skills, and abilities) that affect students' programming are as follows.

- Prior programming experience (Byrne & Lyons, 2001): Students with prior experience in programming are likely to outperform students with limited experience.
- Aptitude in mathematics (Byrne & Lyons, 2001; White & Sivitanides, 2003) and science subjects (Byrne & Lyons, 2001): Subjects who have math and science aptitude are more likely to have programming skill.
- Intrinsic motivation and comfort level (Bergin & Reilly, 2005): Students who are more intrinsically motivated than extrinsically motivated perform better. Furthermore, the level of intrinsic motivation appears to have a positive effect on performance, with higher intrinsic motivation resulting in better programming results. Students with higher scores on the self-esteem scale will perform better in programming than students with lower scores.

These factors affect students' programming success or skill and thus could influence how they manifest programming behavior while solving a programming problem. We expect that as prior experience, aptitude in mathematics/science subjects, motivation, and self-esteem grow, the student's programming behavior will change to incremental development with less propensity for struggling behavior.

Where We Measure It: The Task Model

Characteristic Features of the Task

The task was to write a program for a series of problems in the programming domain of interest. The program that a student writes for a given problem was analyzed in an environment that can (a) evaluate the program code against a suite of tests designed for that problem and (b) provide instant feedback to the student about the correctness of his/her program. To understand how a student develops code for a given problem, this environment should allow for resubmission of a program or should automatically capture the intermediate steps of program development at certain time intervals or, for instance, each time student saves a code.

An example of the task was for students to write a program that receives as input two numbers from a user and then prints out the larger of the two. We refer to this as the *Bigger Number problem*. The student's program was analyzed via three tests that verified that the output was right when the first number was smaller than the second (Test 1), the output was right when the second number was smaller than the first (Test 2), and the student did not print anything unnecessary (Test 3).

Variable Features of the Task

The environment keeps track of student program development via snapshots that are recorded every time the student saves the code, runs the code, runs tests on the code, or submits the code for grading. Snapshots can be captured in a coarse-grained or fine-grained manner. Coarse-grained capturing takes into account the code only at submission, that is, at the final step when the student has a code for the problem and wants to test it. Fine-grained capturing, on the other hand, requires the student to write code in an integrated development environment or platform that is able to track the intermediate steps of development. An example of fine-grained capturing is implemented in a service called Test My Code (Vihavainen, Vikberg, Luukkainen, & Pärtel, 2013). Test My Code contains a NetBeans plug-in that automatically downloads the exercises to the student’s machine, has the capability to gather snapshots from the students’ programming process, and provides a backend server for assessing students’ work. This plug-in records snapshots (time, code changes) every time the student saves code, runs code, runs tests on the code, or submits the code for grading.

An example of snapshots of a typical student is in Figure 1 for the Bigger Number program. The student first wrote a program (Figure 1a) that passed Test 1 Test 2 when the first number was smaller than the second or vice versa. However, it did not pass Test 3 because it printed additional information when the second number was smaller than the first. After receiving this feedback, the student expanded the code by adding the “else if” statement (Figure 1b). Then the program also passed Test 3 because it did not print any unnecessary outputs when the numbers differed.

```
import java.util.Scanner;
public class BiggerNumber {
   public static void main(String[] args) {
       Scanner input = new Scanner(System.in);
       System.out.println("Type a number:  ");
       int firstNumber = Integer.parseInt(input.nextLine());
       System.out.println("Type another number:  ");
       int secondNumber = Integer.parseInt(input.nextLine());
       if (firstNumber > secondNumber)
          System.out.println("\nThe bigger number of the
                          two numbers given was: " + firstNumber);
       if (firstNumber < secondNumber)
          System.out.println("\nThe bigger number of the
                      two numbers given was: " + secondNumber);
       else
          System.out.println(“\nNumbers were equal: ");
   }
}
```

(a)

```
import java.util.Scanner;
public class BiggerNumber {
   public static void main(String[] args) {
       Scanner input = new Scanner(System.in);
       System.out.println("Type a number:  ");
       int firstNumber = Integer.parseInt(input.nextLine());
       System.out.println("Type another number:  ");
       int secondNumber = Integer.parseInt(input.nextLine());
       if (firstNumber > secondNumber)
          System.out.println("\nThe bigger number of the
                          two numbers given was: " + firstNumber);
      else if (firstNumber < secondNumber)
          System.out.println("\nThe bigger number of the
                      two numbers given was: " + secondNumber);
      else
          System.out.println(“\nNumbers were equal: ");
   }
}
```

(b)

Figure 1 - ‘Bigger Number’ program of a student: (a) first snapshot, (b) second snapshot

Each problem in the task can have different levels of complexity, indicating its intrinsic difficulty. The levels of difficulty could alternate between easy, moderate, and complex. Intrinsic difficulty can be measured as a function of concepts required for the program to be correct or be based on the student's perception of the difficulty of the task. Another variable feature of the task is whether the student can complete an initial code or whether he or she can start writing the program from scratch. Thus, the initial state of the program could vary from an empty code to a code with an initial skeleton that student has to complete.

Potential Task Products

The product of the task is the program that student has implemented for the problem. A number of features are related to the program in each snapshot and thus could be measured. These feature are (1) whether the code is compliable or not, (2) correctness of the student's program, (3) number of code lines, (3) time interval from the preceding snapshot, (4) whether the student has requested additional support (such as hint) or accessed instructional resources while working on the code, (5) time that student has spent on writing the code, (6) and programming concepts the student used in the code.

How We Measure It: The Evidence Model

Potential Observations

To identify students' programming behavior, as proposed in Hosseini et al. (2014), the correctness and concepts of the code in each snapshot need to be determined. Correctness is determined by compiling the code, running it against the suite of task-specific tests, and recording the ratio of tests that were passed. For example, for the program in Figure1a, the ratio of passed tests is 2/3 (about 0.67) because the program passed only two of the three tests. To obtain programming concepts, the program code should be parsed using domain parsers. As of now, parsers are available for Java (Hosseini & Brusilovsky, 2013) and Python. An example of concepts in the snapshot code is shown in Table 1. Each concept represents a node in the ontology of the domain.

Table 1 - Distinct concepts extracted by JavaParser for snapshot (a) and (b) of the program in Figure 1. Differences between snapshots are in boldface.

Snapshot	Extracted Concepts
(a)	ActualMethodParameter, MethodDefinition, ObjectCreationStatement, ObjectMethodInvocation, PublicClassSpecifier, PublicMethodSpecifier, StaticMethodSpecifier, StringAddition, StringDataType, StringLiteral, LessExpression, java.lang.System.out.println, java.lang.System.out.print, ClassDefinition, ConstructorCall, FormalMethodParameter, GreaterExpression, **IfElseStatement**, **IfStatement**, ImportStatement, IntDataType, java.lang.Integer.parseInt, VoidDataType
(b)	ActualMethodParameter, MethodDefinition, ObjectCreationStatement, ObjectMethodInvocation, PublicClassSpecifier, PublicMethodSpecifier, StaticMethodSpecifier, StringAddition, StringDataType, StringLiteral, LessExpression, java.lang.System.out.println, java.lang.System.out.print, ClassDefinition, ConstructorCall, FormalMethodParameter, GreaterExpression, **IfElseIfStatement**, **IfElseStatement**, ImportStatement, IntDataType, java.lang.Integer.parseInt, VoidDataType

After determining the concepts and correctness of each snapshot for a single user in each problem that user has solved, conceptual differences between consecutive snapshots need to be examined—i.e., which concepts were added or removed on each snapshot and how these changes were associated with improving or lessening the correctness of program.

Potential Frameworks

To identify students' programming behavior, submission data can be mined, particularly by looking into sequences of concept changes in the programs as well as respective changes in the correctness of programs. Based on the framework introduced in Hosseini et al. (2014), four major problem solving behavior types are defined for the program writing task: (1) Builders, who tend to incrementally build the program and improve its correctness; (2) Reducers, who tend to reduce the concepts and increase correctness; (3) Massagers, who have long streaks of making small program changes without changing concepts or correctness; and (4) Strugglers, who spend considerable time attempting to pass the first correctness test; they make all kinds of code changes but probably have too little knowledge to get the code working.

Table 2 shows how sequences of snapshots are labeled on the basis of the change of concept and correctness in each snapshot.

Table 2- Labelling sequence of students' snapshots based on change of concept and correctness

Correctness/Concepts	Same	Increase	Decrease
Zero	Struggler	Struggler	Struggler
Decrease	Struggler	Struggler	Struggler
Increase	Builder	Builder	Reducer
Same	Massager	Builder	Reducer

After all snapshots have been labeled, the programming behavior of the student can be defined on the basis of a single assignment (local behavior) or over all the assignments the student worked on (global behavior). In either case, we define a profile for a student's programming behavior. The profile records the ratio of occurrence of each of the programming behaviors (Builder, Reducer, Struggler, Massager) in the snapshots of a single assignment for the student's local behavior or over all assignments' snapshots for the student's global behavior. The programming behavior of the student is his or her most frequent behavior— the behavior with the highest ratio of occurrence in the student's profile.

These programming behaviors are related to the classification of Perkins et al. for novice problem solving behavior that classifies students as "stoppers," "movers," or "tinkerers" based on the strategy they choose when facing a problem (Perkins, Hancock, Hobbs, Martin, & Simmons, 1986). Relative to the findings of Hosseini et al. (2014), Builders roughly correspond to "movers"; they gradually add concepts to the solution while increasing the correctness of the solution in each snapshot. Massagers and Reducers could be considered as a mixture of movers and tinkerers. Strugglers, on the other hand, could be considered as a mix of tinkerers and stoppers; they may freeze on a problem (not going toward the correct solution) for a long while, but through experimentation and movement typically end up getting the solution right.

Discussion and Future Work

This paper is an attempt to understand how students develop programs in their assignments or programming exercises. We used ECD to describe a data-driven approach for identifying students' programming behavior by analyzing their program construction activities. This approach examined the conceptual differences between consecutive steps in programming and how these changes are associated with improving or lessening the correctness of the program.

Our past work using this approach (Hosseini et al., 2014) indicated that for the majority of students (77.63%) the dominant pattern was building, incrementally enhancing the program while passing more and more tests. A number of students did it in a very different way, though. Few students tended to reduce the concepts and increase correctness (2.05%) or had long streaks of small program changes without changing concepts or correctness level (0.12%). Some students (20.2%) displayed different programming behavior, showing two or more patterns (e.g., struggling-building, struggling-building-massaging), hinting that their behavior could change over time.

These results point to the importance of distinguishing differences in students' programming behaviors. Although building was the dominant behavior of a large fraction of students, some students tended to develop programs differently. We believe that the differences between students' programming behavior can be used to determine when to provide remedial help for students who struggle while developing a program. Instructors can use this information to understand concepts that students have trouble with. We also believe that students' programming behavior can be a useful source for accurate modeling of students' knowledge.

For future work, we are interested in studying how information about students' programming behaviors could be leveraged to increase the accuracy of student models. Other future work could be to investigate how information about traits that have been used to predict programming aptitude (such as math performance), as well as task products that were not studied in the current approach (such as time spent on writing a code, time interval between snapshots, etc.), influence programming behavior.

Appendix

Author	
First Name	Roya
Last Name	Hosseini
Affiliation	Intelligent Systems Program University of Pittsburgh
E-Mail	roh38@pitt.edu

Overview	
Summary	*Briefly describe the construct, learning environment, and data used.* The construct addressed in this design framework is problem solving in the domain of programming. We analyzed students' program construction activities to determine how students solve programming problems, how students progress to the correct solution through several incorrect ones, and how students' knowledge grows from assignment to assignment. We examined conceptual differences between consecutive snapshots and how these changes were associated with improving or lessening program correctness. The program that students wrote for a given problem were analyzed in an environment that can (a) evaluate the program code against a suite of tests designed for that problem and (b) provide the student with instant feedback about the correctness of the program. To understand how a student develops code for a given problem, this environment should allow resubmission of a program or it should automatically capture intermediate steps of program development at certain time intervals, such as each time the student saves a code.
	Provide seminal citations or papers on the non-cognitive construct, environment, and/or data. • Hosseini, R., & Brusilovsky, P. (2013). JavaParser: A fine-grained concept indexing tool for Java problems. In Proceedings of the 1st Workshop on AI-supported Education for Computer Science (AIEDCS), 60-63. • Vihavainen, A., Vikberg, T., Luukkainen, M., & Pärtel, M. (2013, July). Scaffolding students' learning using Test My Code. In Proceedings of the 18th ACM Conference on Innovation and Technology in Computer Science Education, 117-122. ACM. • Hosseini, R., Vihavainen, A., & Brusilovsky, P. (2014). Exploring problem solving paths in a Java programming course. In Proceedings of the 25th Workshop of the Psychology of Programming Interest Group, 65-76.
Rationale	*Describe the overall importance of the construct being measured.* The research on automatic assessment of programming assignments has led to positive outcomes for instructors and students. Many teams developed automated systems that enabled assessment of large volumes of assignments without requiring much manual effort (Benford et al., 1993; Edwards et al., 2008). This has resulted in changes in how programming is taught. It has also enabled instructors to increase the number of programming assignments in a regular course and offer multiple submission attempts for each assignment (Douce et al., 2005). Automatic assessment systems accumulate a large number of program submissions that in turn open a window on students' problem solving process. Yet researchers are discussing much more rarely automatic assessment systems' important ability to increase our understanding of how humans solve programming problems. Submissions for a single problem have been studied to discover multiple correct and incorrect ways to solve the problem (Huang et al., 2013; Piech et al., 2012) or to build an intelligent scaffolding system (Rivers & Koedinger, 2013). In the present work, we studied the less explored topic related to automatic assessment systems. We describe a data-driven approach to using students' submission data to understand how students solve their programming assignments and to identify their programming behaviors.

	For what purpose(s) will claims or inferences related to the construct be used? Identification of programming behaviors using observational data collected from students' problem solving activities is beneficial for both instructors and students. • It enables instructors to better address concepts that students find difficult to grasp or are having misconceptions about (Tabanao et al., 2011). • It helps identify and provide targeted help and proper intervention for students who are strugglers. • Identifying student programming behavior may help mitigate student frustration and confusion, increase programming comprehension, and contribute to greater student retention in a course (Tabanao et al., 2011). This information can also be used to provide proactive recommendations and suggestions to all students, for instance, on how long to expect an assignment to take or what errors to look out for (Murphy et al., 2009).

Student Model	
Focal construct	*Name the primary construct addressed by this design pattern.* The focal construct addressed in this design framework is problem solving in the domain of programming.
Additional knowledge, skills, and abilities	*Identify sources of construct irrelevant variance or confounds (i.e., other knowledge, skills, or abilities) that may affect how students manifest a construct, data quality, or measurement.* • Prior programming experience (Byrne & Lyons, 2001) • Intrinsic motivation and comfort level (Bergin & Reilly, 2005) • Aptitude in mathematics (Byrne & Lyons, 2001; White & Sivitanides, 2003) and science subjects (Byrne & Lyons, 2001)

Task Model	
Characteristic features of the task	*Aspects of the task or task environment that are required to evoke evidence about the focal construct* • The task is to write a program for a series of programming problems in the programming domain of interest. In this paper we focus on Java programming, but the approach could be used for other programming domains, too. • The task environment should have the following features: ○ allow resubmission of a program ○ capture snapshots of a student's program (snapshots recorded every time the student saves code, runs code, runs tests on the code, or submits the code for grading ○ can evaluate correctness of the student's program, perhaps by running unit tests, and provide feedback to the student, accordingly.

Variable features of the task	*Aspects of the task or task environment that can vary, or can be intentionally varied, to affect how students enact the focal construct* • Snapshots of the student's program can be captured in a coarse-grained or fine-grained manner: A coarse-grained capturing takes into account only the code at submission time, the final step when student has a code for the problem and wants to test it. A fine-grained capturing, on the other hand, requires the student to write code in an integrated development environment or platform that is able to track intermediate steps of program development. • Problem complexity, indicating intrinsic difficulty. • The initial state of the program, which could vary from empty code to a code with an initial skeleton that student has to complete.
Potential task products	*That which students say, do, or make that produces or contains evidence of the focal construct.* The product of the task is the program the student has implemented for the problem. There are number of features related to the program in each snapshot, including: • whether the code is compliable or not • correctness of student's program • number of code lines • time interval from preceding snapshot • whether student has requested additional support (such as hint) or accessed instructional resources while working on the code • time that student has spent writing the code and programming concepts student used in the code.

Evidence Model	
Potential observations	*Qualities of the potential task products (e.g., excessive, limited, or correct) that can be used to make inferences about focal construct.* To identify student's programming behavior, as proposed in (Hosseini et al., 2014), we determined the following features for the program in each snapshot: • correctness of the program, obtained by compiling the code, running it against the suite of task-specific tests, and recording ratio of tests that were passed. • concepts that student used in the program ○ For obtaining programming concepts, the program code should be parsed using domain parsers. An example of parser for Java is in Hosseini & Brusilovsky (2013). • conceptual differences between consecutive snapshots – which concepts were added to or removed from each snapshot relative to the previous snapshot and how these changes were associated with improving or lessening the correctness of the program.
Potential frameworks	*Potential frameworks (e.g., rubrics, algorithms, rules) used to interpret, judge, or contextualize potential observations* Based on the framework introduced in Hosseini et al. (2014), four major problem solving behavior types were defined for the programs writing task: (1) Builders, who tend to incrementally build the program and improve its correctness; (2) Reducers, who tend to reduce the concepts and increase correctness; (3) Massagers, who have long streaks of making small program changes without changing concepts or correctness; and (4) Strugglers, who spend considerable time attempting to pass the first correctness test; they do all kinds of code changes but probably have too little knowledge to get the code working. Thus, to identify students' programming behavior, submission data can be mined, particularly by looking into (a) sequences of concept changes in the programs and (b) respective changes on the correctness of programs. The following table shows how sequences of snapshots are labeled based on

the change of concept and correctness in each snapshot.

Correctness\Concepts	Same	Increase	Decrease
Zero	Struggler	Struggler	Struggler
Decrease	Struggler	Struggler	Struggler
Increase	Builder	Builder	Reducer
Same	Massager	Builder	Reducer

After all snapshots have been labeled, the programming behavior of the student can be defined on the level of a single assignment (local behavior) or over all assignments student worked on (global behavior). In either case, we define a profile for student's programming behavior. The profile records the ratio of occurrence of each of the programming behaviors (Builder, Reducer, Struggler, Massager) in the snapshots of a single assignment for the student's local behavior or over all assignments' snapshots for the student's global behavior. The programming behavior of the student is his or her most frequent behavior—the behavior with the highest ratio of occurrence in the student's profile.

References

Benford, S., Burke, E., & Foxley, E. (1993). Learning to construct quality software with the Ceilidh system. Software Quality Journal, 2(3), 177-197.

Bergin, S., & Reilly, R. (2005). The influence of motivation and comfort-level on learning to program. In Proceedings of the 17th Workshop of the Psychology of Programming Interest Group, 293–304.

Blikstein, P. (2011, February). Using learning analytics to assess students' behavior in open-ended programming tasks. In Proceedings of the 1st International Conference on Learning Analytics and Knowledge, 110-116. ACM.

Byrne, P., & Lyons, G. (2001, June). The effect of student attributes on success in programming. In ACM SIGCSE Bulletin, 33(3), 49-52. ACM.

Douce, C., Livingstone, D., & Orwell, J. (2005). Automatic test-based assessment of programming: A review. Journal on Educational Resources in Computing (JERIC), 5(3), 4, 1-13. ACM.

Edwards, S. H., & Perez-Quinones, M. A. (2008, June). Web-CAT: automatically grading programming assignments. In ACM SIGCSE Bulletin, 40(3), 328-328. ACM.

Hosseini, R., & Brusilovsky, P. (2013). JavaParser: A fine-grained concept indexing tool for Java problems. In Proceedings of the 1st Workshop on AI-supported Education for Computer Science (AIEDCS), 60-63.

Hosseini, R., Vihavainen, A., & Brusilovsky, P. (2014). Exploring problem solving paths in a Java programming course. In Proceedings of the 25th Workshop of the Psychology of Programming Interest Group, 65-76.

Huang, J., Piech, C., Nguyen, A., & Guibas, L. (2013, June). Syntactic and functional variability of a million code submissions in a machine learning mooc. In AIED 2013 Workshops Proceedings Volume, 25-32.

Murphy, C., Kaiser, G., Loveland, K., & Hasan, S. (2009). Retina: Helping students and instructors based on observed programming activities. ACM SIGCSE Bulletin, 41(1), 178-182. ACM.

Perkins, D. N., Hancock, C., Hobbs, R., Martin, F., & Simmons, R. (1986). Conditions of learning in novice programmers. Journal of Educational Computing Research, 2(1), 37-55.

Piech, C., Sahami, M., Koller, D., Cooper, S., & Blikstein, P. (2012, February). Modeling how students learn to program. In Proceedings of the 43rd ACM technical symposium on Computer Science Education, 153-160. ACM.

Rivers, K., & Koedinger, K. R. (2013, June). Automatic generation of programming feedback: A data-driven approach. In Proceedings of the 1st Workshop on AI-supported Education for Computer Science (AIEDCS), 50-59.

Rowan, T. C. (1957). Psychological Tests and Selection of Computer Programmers. Journal of the Association for Computing Machinery, 4, 348-353.

Tabanao, E. S., Rodrigo, M. M. T., & Jadud, M. C. (2011, August). Predicting at-risk novice Java programmers through the analysis of online protocols. In Proceedings of the 7th International Workshop on Computing Education Research, 85-92. ACM.

Thomas, L., Ratcliffe, M., Woodbury, J., & Jarman, E. (2002). Learning styles and performance in the introductory programming sequence. ACM SIGCSE Bulletin, 34(1), 33-37. ACM.

Vihavainen, A. (2013, July). Predicting students' performance in an introductory programming course using data from students' own programming process. In 2013 IEEE 13th International Conference on Advanced Learning Technologies (ICALT), 498-499. IEEE.

Vihavainen, A., Luukkainen, M., & Kurhila, J. (2013, July). Using students' programming behavior to predict success in an introductory mathematics course. In Proceedings of the 6th International Conference on Educational Data Mining, 300-303.

Vihavainen, A., Vikberg, T., Luukkainen, M., & Pärtel, M. (2013, July). Scaffolding students' learning using Test My Code. In Proceedings of the 18th ACM Conference on Innovation and Technology in Computer Science Education, 117-122. ACM.

Watson, C., Li, F. W., & Godwin, J. L. (2014, March). No tests required: Comparing traditional and dynamic predictors of programming success. In Proceedings of the 45th ACM Technical Symposium on Computer Science Education, 469-474. ACM.

White, G., & Sivitanides, M. (2003). An empirical investigation of the relationship between success in mathematics and visual programming courses. Journal of Information Systems Education, 14(4), 409-416.

Yudelson, M., Hosseini, R., Vihavainen, A., & Brusilovsky, P. (2014, July). Investigating automated student modeling in a Java MOOC. In Proceedings of the 7th International Conference on Educational Data Mining, 261-264