

Planning and the Novice Programmer: How Grounded Theory Research Can Lead to Better Interventions

Jonathan Wellons, Julie Johnson

Department of Electrical Engineering and Computer Science
Vanderbilt University

Email: {jonathan.wellons, julie.l.johnson}@vanderbilt.edu

Abstract—Planning is a critical, early step on the path to successful program writing and a skill that is often lacking in novice programmers. As practitioners we are continually searching for or creating interventions to help our students, particularly those who struggle in the early stages of their computer science education. In this paper we report on our ongoing research of novice programming skills that utilizes the qualitative research method of grounded theory to develop theories and inform the construction of these interventions. We describe how grounded theory, a popular research method in the social sciences since the 1960’s, can lend formality and structure to the common practice of simply asking students what they did and why they did it. Further, we aim to inform the reader not only about our emerging theories on interventions for planning but also how they might collect and analyze their own data in this and other areas that trouble novice programmers. In this way those who lecture and design CS1 interventions can do so from a more informed perspective.

Index Terms—Novice Programmers, Planning, Qualitative Research, Grounded Theory

I. INTRODUCTION

There is much research in the area of self-regulated learning and its effects on student performance. Students who report exercising such skills as goal setting, planning, self-monitoring and self-evaluation experience higher levels of success and satisfaction than students who do not [1]. For programmers, planning is one of the first critical self-regulatory skills they will need. Early programming experiences are defined by a novice’s ability to engage the complex cognitive process of problem solving while employing the metacognitive processes of self-regulation. While advice in teaching problem solving abounds, only recently have we seen developments in software tools and learning modules that address the process of planning. The overall goal of our research is to design activities and scaffolds to teach and support the metacognitive skills that novice programmers need to achieve early success in programming. The focus of this qualitative study was to use a systematic approach to observe and explain the process of planning among novice programmers at an undergraduate university. We began by analyzing interview data with the aim of developing emergent theories. Such theories and the subsequent results of testing (not a part of this study) will lead to the development of tools to help students improve and refine this critical self-regulatory skill. More specifically we asked, “What is the theory that explains the process of planning among novice programmers with no formal instruction in making such plans?” and “How could such a theory inform

the construction of scaffolds and learning tools for novice programmers?”.

In order to address these questions we employ a grounded theory study. This methodology is common in the social sciences and is appropriate when attempting to develop the groundwork for theories while avoiding pre-existing biases. Our objective is to produce data which can be qualitatively examined for connections between novice programming planning, causes or effects of their planning skills and other habits or tendencies on the part of the students.

II. THE ROLE OF PLANNING IN THE PROGRAMMING PROCESS

Developers use plans in large-scale projects to model a program at a manageable level of abstraction [2]. It is universally accepted that programming successfully requires planning and many different strategies are currently in use. The general model first put into wide use is the waterfall, consisting of distinct stages of requirements, design, implementation, testing and maintenance [3]. More flexible models gradually came to be applied to software beginning with the idea of iterative and incremental development, a cyclic process that applies elements of the waterfall model multiple times allowing for adaptation [4]. In the past few decades many of the same principles reappeared in slightly different forms, such as the spiral model [5] and more recently extreme programming, agile programming and test-driven development. Extreme programming is characterized by frequent release cycles, pair programming, extensive unit testing and flexible schedules [6], [7]. Agile programming emphasizes open and frequent communication, adaptability, customer stakeholders and cross-functional teams [8], [9]. The principle of test-driven development is to first institute a unit test before coding each feature or bug fix [10]. Numerous other ideas have been practiced such as cleanroom software engineering which was developed to provide reliable and verifiable software [11] and lean software development which aims to eliminate waste in all forms (excess bureaucracy, requirements, code, delay, etc.). It is clear that in order to participate in the design, management, development and testing of large-scale projects, students must develop the requisite planning skills exhibited by productive computer scientists and engineers.

III. RESEARCH ON PLANNING

In his 1986 article Soloway called for a redesign of Computer Science curriculum to include the explicit teaching of

problem solving skills which included planning and goal setting. He noted that expert programmers drew from a library of canned solutions to form a template or starting point for their solutions. Soloway proposed using goal/plan language for teaching introductory programming thus making the role of plans and goals more explicit to the novice from the outset [12]. In [13] it was demonstrated that teaching programming strategies and planning was possible, did not increase time needed for instruction and could be measured through written assessment.

In response to these reports, several tools have been designed and tested in an effort to support planning and strategy selection using the expert programmer's approach as a model. In [14] 25 undergraduate students in two randomly assigned groups received training in planning by one of two methods. The treatment group used an intelligent tutoring system (ProPL) to implicitly scaffold planning through the use of prompts while the other group received "click-through" text describing planning and strategies for programming. Their results demonstrated the value of a scaffolded approach to teaching planning skills as the ProPL group fared better in the assigned programming tasks than did the other group. In [15], [16] the authors describe two object oriented programming languages (Visual Plan Construct Language and Web Plan Object Language) designed to teach programming to novices through plan management and integration. These languages facilitate the Plan-Object Paradigm, an approach that gives context to programming objects by allowing students to create a plan first and then use that plan to create working programs.

In a more broad approach, [17] explored scaffolds for scientific inquiry and the needs of learners engaging in new and complex work processes, an appropriate description for novice programming. The authors evaluated Symphony, a software tool meant to scaffold the learner's planning activities. They used the artifacts created by the tool to analyze the complex process of scientific inquiry, identifying the needs of learners that could be further scaffolded. In this way, the tool itself became a means by which further research could be pursued.

Despite these advances in supporting the novice programmer, seventeen years after Soloway's article, [18] reported that strategies and plans, though crucial to learning outcomes in introductory programming courses still receive much less attention than language related knowledge. They also noted that the questions of why and how different strategies emerge and how they are related to underlying knowledge was still an open question.

Introductory programming classes differ in their emphasis on planning. Those that encourage it often do so in different forms. One class may teach the writing of program comments in advance of any code and a different class might teach top-down modularization. Students exposed to a limited style of planning or not exposed to planning at all may experience deceptive initial success with small projects but have difficulties later. Basic programming tasks are notoriously difficult for students to master and many potential majors leave computer science or fail to gain essential skills (examples from Australia, the US and the UK in [19], [20]). Given that programming is an essential skill in many engineering disciplines (many

with a shortage of employees) as well as advanced coursework and that an enormous diversity of educational approaches are employed, it is natural to ask if there is a better way to teach planning.

Kuhl and Goschke [21] proposed a model for self-regulated learning which includes the steps of goal setting and planning. Their model was recursive; learners would return to the tasks again and again as they experienced internal feedback from the products they generated. This recursion is made more overt in the programming experience as learners receive explicit feedback from the compiler or debugger used while building a program. Error messages and programs that fail to terminate are external signs to the learner that a change is required. With few experiences to draw from, reflection is often limited to the code itself and incremental changes are made in an effort to improve the outcome. Changes often lead to incremental success which in turn contributes to progress. When progress leads to program completion this abbreviated reflective cycle, fully contained within the coding exercise, becomes a template for future programming assignments. As programming problems become more complex progress may be significantly slowed or halted signaling the learner that reflection on the larger plan may be in order. Whether this indication is taken up or ignored, in the presence or absence of an initial plan constructed by the novice, may yield some insight into the initial planning process.

IV. THE USE OF GROUNDED THEORY AS A METHOD OF ANALYSIS

Grounded theory is a form of qualitative research based on the formation of theory from data. Open-ended interview questions are posed and data is collected in an effort to generate theories about the domain in question. This methodology can be described in five steps: 1) gathering of data, 2) open coding whereby researchers assign discrete codes to the qualitative data, 3) grouping of codes into concepts and identifying one or more important concepts that merit reexamination of existing data and possibly further data collection to facilitate the building of a model, 4) axial coding, meaning the construction of categories that highlight the relationship between concepts and 5) the suggestion of one or more theories which describe the relationships. Grounded theory researchers cast a wide net to capture a diverse and multi-dimensional data set which may be fertile ground for theories [22]. Making every attempt to avoid the influence of prior theories or other constructs, researchers allow the data to form the theory rather than using existing theories to code and categorize the data.

As practitioners in the university system, we routinely gather data from our students in an effort to gauge both the effectiveness of our teaching and the impression it leaves on the students themselves. Performance data coupled with course evaluations serve as a reliable record of the course outcomes. Open-ended questions are often posed to students in an effort to collect qualitative data to inform future modifications to the course ("What if anything would you change about this course?") or to elicit reaction to recent modifications ("How did you use the online planning modules when completing

programming assignments?”). When we use this data to make course refinements we are using some of the techniques formalized in grounded theory.

We synthesize this data with performance data, anecdotal evidence and past experience to guide our subsequent steps. This type of approach is the basis for such research methods as design based research or action research. By contrast, grounded theory formalizes the process of data analysis and produces a theory or set of theories that can then be used to develop course modifications, controlled experiments or future research in a broader context.

Qualitative research has been used to investigate novice programming, although grounded theory has not been applied to planning specifically. Interviews and coding are used in [23] to investigate how non-major programmers conceptualize Java concepts. To investigate if some elementary programming tasks are more difficult than others, [24] searched for novice programmer bottlenecks in object oriented programming. Students were observed during labs and their affective states and behaviors were coded. Compiler error logs coupled with interviews were used in [25] to track the most common errors made by novice programmers.

Like those involved in the BRACElet Project and the others, we believe that challenges to CS curriculum should be approached as research problems requiring established research methods [26]. We chose to use the grounded theory approach for several reasons. First, we wanted to use a method of data collection that was already familiar to us and to many practitioners in the field of computer science education and would not require additional equipment or instrumentation. Second, the systematic approach to data analysis appealed to us as computer science educators, acting as a segue to other more unfamiliar qualitative research methods. And finally, we wanted to demonstrate how student interviews could be used for more than just the course on which they were commenting, but could yield insights into more general aspects of the programming experience.

V. METHOD

Our sample consisted of volunteers from three sections of an Introduction to Engineering class offered at a research university in the United States. The course, required for all freshmen engineering majors, was presented in one four-week module consisting of approximately 14 one-hour lectures. This course introduces the fundamentals of programming (using MATLAB) within the context of cryptography. The general idea behind the course is to familiarize students with problem solving techniques and tools (such as MATLAB and Excel) while giving them a snapshot of the field of Computer Science and some of its practical applications. The volunteers came from varying backgrounds and were not all necessarily declared Computer Science majors. They were compensated for their time and reflected approximately the same gender breakdown as the class. None who volunteered were denied inclusion in the study. Because the goal of initial data collection is to gather as many different stories and experiences as possible, thus saturating each category with

explanations and examples, random sampling is not as critical in grounded theory. In fact, in our discussion section we describe future data collection that will involve theoretical sampling—the selection of data based on the potential to represent the core theoretical constructs being studied.

In order to gather accurate data, volunteers’ names were removed from their interviews. We asked open-ended questions and students were encouraged to discuss any aspects of their programming experience that they deemed meaningful. The interviewer gathering the data, held office hours and guest lectured in two or three class sessions of each of the three sections to develop a rapport with students and introduce the study. All interviews were voice recorded for later analysis. Volunteers were encouraged to describe their experiences during several assignments, what type of plan they created, how detailed it was, and how it was adapted. Subjects were questioned about their background in programming, their hobbies and other details that could lead to a planning theory. Several hundred codes were derived from the data, which were in turn placed into 14 concepts. Finally, we organized the concepts into five categories which naturally suggested the three theories that comprise our results, as detailed in Sec. VI.

VI. RESULTS

Concepts are derived organically, based only on the codes that are present. The interviews explicitly asked for the content and complexity of the student’s plan. Examples include “I made a list of tasks,” “I just started writing” and “I wrote the comments then filled in the code.” These are clustered in the *Initial Plan* concept. In addition, we asked how the plan performed when the student attempted to implement it and whether the plan evolved. One subject reported “it all fell in pretty nicely.” Another said “I realized I didn’t need some things. I had difficulty making the alphabet [substitution cipher] work.” These are coded in the *Plan Adequacy* concept.

We invited students to describe their programming process. One subject reported “I had a hard time keeping track of variables for rows, columns, indices and so on.” Another subject “... relied on old programs and examples.” Other students reported difficult language features and the mechanical details of their run-debug cycle. These codes are grouped into the *Coding Process* concept. Subjects were asked about sources of help during the programming assignments. Many reported seeking aid from classmates and friends who were advanced engineering majors or had previous programming experience. Many students used Google, the MATLAB online or built-in help, the professor or the textbook. These codes are clustered into the *Help Sources* concept. All subjects are asked about their debugging process. One student said that after working on a frustrating bug “I took a day off to clear my mind, then returned to see if it was right.” Some students reran their program after every newly added line to check for errors. Others ran it only when they believe the programming was finished. A handful of students used debugging output statements. Considerable diversity was present in the types of test cases used. Some students used only the cases given in the assignment, others gave random test cases. These codes are in the *Testing* concept.

TABLE I
EXAMPLE CODES AND THE CORRESPONDING CONCEPT

Sample Code	Assigned Concept
Initial plan was a short, vague list of tasks	Initial Plan
Plan failed because MATLAB doesn't handle long strings	Plan Adequacy
Program was constructed of fragments adapted from in-class examples	Coding Process
Searched for help on Google, e.g., "how to write a for loop"	Help Sources
Wrote program in 3-4 parts which were tested separately and combined at the end	Testing
"My goal was to receive a B or an A"	Goals
It took 2 to 2.5 hours to finish. Student had thought it would take 1 hour	Time Needed
"First day was too much, second day I started to understand, it clicked on the third day"	Class Experience
Programming background consists of using the Starcraft map editor	Programming Background
Had some advanced Math, "not good at it" but "loves it"	Quantitative Background
Calls self a "number cruncher" in everyday life	Hobbies
"Why go for the extra credit when I don't understand the basics"	Ambition
Rather than write a time-consuming brute-force program, he solved the permutation puzzle visually out of 362,880 possibilities	Lateral Thinking
Influenced lab partners to use pseudocode in the future	Personality

Subjects were asked about the type of goals they set for the assignment. Some students focused on grades, one reported "a B or an A." Another said simply "to finish." Other goals were "to finish before the weekend" and "initially I just wanted it to work, but later I wanted to satisfy my intellectual curiosity." Several students reported they enjoyed the assignment and no external motivation was necessary, but only one reported that learning MATLAB was his goal. These codes were placed in the *Goals* concept. Student were asked how much time they took to complete the assignment. Most responses were between 1.5 and 3 hours. Subjects typically found it took longer than they had expected, although there were exceptions. All codes related to how long the assignment took and how this compared to the student's prior expectation were combined in the *Time Needed* concept. Students were prompted for information describing their educational experience in the course which we classified under the *Class Experience* concept. One reported, "the scavenger hunt [assignment] was fun because cleverness was required." Another response: "Oh the hash code, that was frustrating!" Other students reported that the class moved too fast or that the examples were not related. A subject said that he was lost on the first day, began to understand on the second day and "it clicked" on the third day. We developed another concept measuring ambition from the reactions to a cryptography assignment which required students to choose between four algorithms with differing degrees of difficulty. Each choice was accompanied by a maximum possible number of points, ranging from 110 for the most complex algorithm to 87 for the most straightforward. Several intermediate choices were also given (such as input restrictions or user interface affordances) that could increase the point value of the attempt. A student's choices could be perceived as a measure of their self-confidence. Many students aimed low. One reported, "why go for the extra credit when I don't understand the basics." Others chose combinations over 100, but not the maximum possible. These codes are placed under the *Ambition* concept.

The interviews included questions about each subject's background in programming to discover a relationship between a student's previous experience and planning or assignment success. Many students in the sample had little or no background in programming. The most extensive backgrounds came from AP classes in Java and toy problems on a educational website. Another subject had learned MATLAB and C++ over the summer. A subject had used a script-based map editor for the game Starcraft. These codes were grouped in the concept of *Programming Background*. Subjects also discussed their background in quantitative studies. Many students reported enjoying and excelling at Math. One said "Math is my best subject." A handful of students were ambivalent about Math, "I went up to AB Calculus because BC was like boot-camp." These codes were clustered in *Quantitative Background*.

Students were asked to describe their hobbies. Many students were interested in strategy, board, card and video games. Only one reported sports. One reported, "Piano, writing poetry and Chess." Guitar playing was mentioned. These codes were grouped in the *Hobbies* concept. In the course of describing their problem solving plans, students often revealed insightful solutions. One problem required decrypting a scrambled message with nine factorial (362,880) possible keys. Rather than write the brute-force program suggested in class, several students were able to crack it with pencil and paper, or use creative shortcuts that reduced the complexity of the program to write. One student solved this puzzle in Excel using built-in functions. We label these codes with the concept *Lateral Thinking*. During the student's description of their problem-solving techniques or group work, aspects of their personalities were revealed. One strong proponent of using pseudocode reported that she had influenced her teammates to use it on the following individual assignments. When codes of this type became apparent, they were classified in the *Personality* concept.

At this phase of the project, 14 concepts emerged as

natural partitions to the codes as shown in Table I. We then transition to the axial coding step and aggregate concepts into categories based on similarity. Five categories emerged as ideal clusterings as shown in Table II. The data organization is bottom-up and is reflected by placing the child data on the left and the parent data on the right.

VII. EMERGING THEORIES

Our goal in grounded theory is to examine the apparent connections between categories to suggest theories which can explain the data before us. The process of selective coding requires the selection of a core category. Connections are then studied in an attempt to define the relationships between all other categories and the chosen core category. From this rich set of codes and concepts, many relationships are possible. For example, connecting the *Planning* and *Programming Methodology* categories we found that plans that included testing related to shorter total time spent on the program. Relationships between *Planning* and *Goal Setting and Achievement* included a connection between those who planned and the extent of their ambitions in the program. Other connections that arose include the relationship between students who use lateral thinking and a lower frustration level during their programming experience. To expand any one of these relationships into a working theory would require theoretical sampling followed by further coding in an effort to saturate the data concerning the categories being related. This exercise would allow us to strengthen the proposed theory.

Ideal theories will not only be supported by the data, but potentially lead to research to enhance pedagogy for novice programmers. In keeping with the conventions of grounded theory, we want to avoid the problem of existing theories or claims influencing our analysis of the data. We do however turn to existing literature to identify gaps that might be informed by our work.

Expert planners have amassed a library of templates that they can draw from and flexibly apply to the problem at hand [12]. Among novice programmers, for whom such a mental library most likely does not exist, we do not know what form the planning process takes. Investigating the data suggests that novice programmers borrow ideas from their areas of relative expertise. Students expressly referred to their math knowledge or experiences writing papers when describing the origin of their plans. This leads us to our first candidate theory:

Theory I: Novice programmers attempt to adapt problem-solving strategies from other domains, such as mathematics or essay composition.

To guide us to the next theory, we investigate how to build a scaffold that can serve in the place of this expert library of templates until it can be established. Based on the data, students who wrote pseudocode were more likely to perceive success on the assignment, have higher ambition and have an adequate plan. This is true regardless of whether the student had a programming background, leading us to consider that the traditional skill of pseudocode may be the only scaffold needed for first-time programmers, at least until they have successfully

completed a few assignments. Furthermore, effectively using pseudocode is teachable which leads to the following theory.

Theory II: Planning by means of pseudocode is achievable for novice programmers.

Equally important to CS1 lecturers is how to develop the feeling of perceived success in novice programmers. Based on various comments about the quality of a student's learning experience which show a relationship between planning and perceived success in the class as a whole, we formulate the final theory:

Theory III: Students who planned their programs are more likely to report a positive experience in the class.

VIII. LIMITATIONS AND FUTURE WORK

Within the scope of grounded theory which is the production rather than the testing of theories, this work has several areas for further development. The students were from a single class intended for students seeking an engineering major. Thus the results do not immediately apply to either majors or non-majors in computer science because it is not known if both of these sets are well represented in the data. Furthermore, the sample of students is somewhat small and self-selected. The next step is theoretical sampling, which deliberately chooses samples in order to diversify the data set. The authors are likely to theoretically sample different backgrounds and expected majors, as well as a class with a different style of teaching and programming language. During theoretical sampling, the original data will be added to rather than replaced.

Practitioners divide research on teaching into two kinds: "What is" and "What works". "What is" research focuses on observations about current conditions and processes in the learning environment. "What works" research tests and measures alternative teaching practices. This is a "What is" project attempting to suggest relationships between novice program planning and other concepts for further study.

Outside of using grounded theory to qualitatively produce theories, future work consists of two kinds: refinement of these candidate theories and the development of "What works" projects to realize the benefits of proving or disproving these theories. Having established three candidate theories, the next phase of our research is to collect more qualitative data to strengthen or deny each of them. The strengthening of Theories I and II could lead to guidelines for developing learning modules and subsequent scaffolding designed specifically for the novice programmer while further exploration of Theory III would naturally lead us to a quantitative study to verify the proposed relationship. Each candidate theory has the potential to affect how we teach the metacognitive skill of planning and the emphasis that we place on that exercise.

IX. CONCLUSION AND SUMMARY

In this paper, we have applied grounded theory to interviews with novice programmers about their first programs. Through the principles of grounded theory we have coded, conceptualized and categorized the interview data. We have elucidated the connections between categories to generate several plausible theories that explain the data. Three candidate theories are

TABLE II
CONCEPTS ASSIGNED TO EACH CATEGORY

Concepts	Assigned Category
Initial Planning, Plan Adequacy	Planning
Coding Process, Help Sources, Testing	Programming Methodology
Goals, Time Needed, Class Experience, Ambition	Goal Setting and Achievement
Programming Background, Quantitative Background	Background
Hobbies, Lateral Thinking, Personality	Personal

proposed in this project, based on the observed relationship between planning strategies and self-reports of programming experience. The theories are, 1) Novice programmers attempt to employ problem-solving strategies from other domains which are more familiar to them, 2) Pseudocode-based planning tends to be a relatively successful strategy for novices and 3) Program planning leads to a positive reported class experience.

X. ACKNOWLEDGMENTS

We gratefully acknowledge the support of the Center for Integration of Research, Teaching and Learning project (NSF Grant No. DUE-0717768) and the Qualitative Research Methods Workshop (NSF Grant No. DUE CCLI 0923592). Without the support of these two projects, this research would not have been possible.

REFERENCES

- [1] B.J. Zimmerman and M. Martinez-Pons, "Development of a Structured Interview for Assessing Student use of Self-Regulated Learning Strategies," *American Educational Research Journal*, vol. 23, pp. 614–628, 1986.
- [2] C. C. Yu and S. P. Robertson, "Plan-based Representations of Pascal and Fortran code," *Proceedings of the SIGCHI*, May 1988.
- [3] Winston W. Royce, "Managing the Development of Large Software Systems," *Proceedings of IEEE WESCON*, pp. 1–9, 1970.
- [4] Larman C. and V.R. Basili, "Iterative and Incremental Developments. a Brief History," *Computer*, vol. 36, no. 6, pp. 47 – 56, June 2003.
- [5] B. Boehm, "A Spiral Model of Software Development and Enhancement," *SIGSOFT Softw. Eng. Notes*, vol. 11, no. 4, pp. 14–24, 1986.
- [6] Mark C. Paulk, "Extreme Programming from a CMM Perspective," *IEEE Software*, vol. 18, pp. 19–26, 2001.
- [7] Kent Beck, *Extreme Programming Explained: Embrace Change*, Addison-Wesley, 2000.
- [8] Agile Alliance, "Manifesto for Agile Software Development," <http://agilemanifesto.org>, 2001.
- [9] Orit Hazzan and Yael Dubinsky, "Why Software Engineering Programs should teach Agile Software Development," *SIGSOFT Softw. Eng. Notes*, vol. 32, no. 2, pp. 1–3, 2007.
- [10] Kent Beck, *Test-Driven Development by Example*, Addison-Wesley, 2003.
- [11] H.D. Mills, M. Dyer, and R.C. Linger, "Cleanroom Software Engineering," *Software, IEEE*, vol. 4, no. 5, pp. 19–25, 1987.
- [12] E. Soloway, "Learning to Program = Learning to Construct Mechanisms and Explanations," *Communications of the ACM*, vol. 29, no. 9, pp. 850–858, 1986.
- [13] Michael de Raadt, Richard Watson, and Mark Toleman, "Teaching and Assessing Programming Strategies Explicitly," *Eleventh Australasian Computing Education Conference (ACE2009)*, vol. 20–23, Jan 2009.
- [14] Kurt Vanlehn and H. Chad Lane, "Teaching the Tacit Knowledge of Programming to Novices with Natural Language Tutoring," *Computer Science Education*, vol. 15, pp. 183–201, 2005.
- [15] A. Ebrahimi and C. Schweikert, "Empirical Study of Novice Programming with Plans and Objects," *SIGCSE Bull.*, vol. 38, no. 4, pp. 52–54, 2006.
- [16] Alireza Ebrahimi, "Novice Programmer Errors: Language Constructs and Plan Composition," *Int. J. Hum.-Comput. Stud.*, vol. 41, no. 4, pp. 457–480, 1994.
- [17] Chris Quintana, Jim Eng, Andrew Carra, Hsin-Kai Wu, and Elliot Soloway, "Symphony: a Case Study in Extending Learner-Centered Design through Process Space Analysis," in *CHI '99: Proceedings of the SIGCHI conference on Human factors in computing systems*, New York, NY, USA, 1999, pp. 473–480, ACM.
- [18] Anthony Robins, Janet Rountree, and Nathan Rountree, "Learning and Teaching Programming: A Review and Discussion," *Computer Science Education*, vol. 13, pp. 137–172, 2003.
- [19] Nghi Truong, Peter Bancroft, and Paul Roe, "A Web Based Environment for Learning to Program," in *ACSC '03: Proceedings of the 26th Australasian computer science conference*, Darlinghurst, Australia, Australia, 2003, pp. 255–264, Australian Computer Society, Inc.
- [20] Michael McCracken, Vicki Almstrum, Danny Diaz, Mark Guzdial, Dianne Hagan, Yifat Ben-David Kolikant, Cary Laxer, Lynda Thomas, Ian Utting, and Tadeusz Wilusz, "A Multi-National, Multi-Institutional Study of Assessment of Programming Skills of First-Year CS Students," in *ITiCSE-WGR '01: Working group reports from ITiCSE on Innovation and technology in computer science education*, New York, NY, USA, 2001, pp. 125–180, ACM.
- [21] J. Kuhl and T. Goschke, *Volition and personality*, Hogrefe and Huber, 1994.
- [22] Patricia Yancey Martin and Barry A. Turner, "Grounded Theory and Organizational Research," *The Journal of Applied Behavioral Science*, vol. 22, no. 2, pp. 141, 1986.
- [23] Anna Eckerdal, Anna Eckerdal, and Anna Eckerdal, "Novice students learning of object-oriented programming," 2006.
- [24] J. O. Sugay M. M. T. Rodrigo, R. S. J. Baker and E. Tabanao, "Monitoring novice programmer affect and behaviors to identify learning bottlenecks," *Philippine Computing Society Congress*, March 2009.
- [25] Suzanne Marie Thompson, "Exploratory study of novice programming experiences and errors," March 2006.
- [26] Tony Clear, Jenny Edwards, Raymond Lister, Beth Simon, Errol Thompson, and Jacqueline Whalley, "The Teaching of Novice Computer Programmers: Bringing the Scholarly-Research Approach to Australia," in *ACE '08: Proceedings of the tenth conference on Australasian computing education*, Darlinghurst, Australia, 2008, pp. 63–68, Australian Computer Society, Inc.