DEVELOPMENT OF A WEB-BASED SURVEYING INSTRUMENT TO IDENTIFY PROBLEM-SOLVING ABILITIES RELATED TO EFFECTIVE INSTRUCTION IN COMPUTER PROGRAMMING

by

Jorge Vasconcelos-Santillán

A dissertation submited to The Johns Hopkins University in conformity with the requirements for the degree of Doctor of Philosophy

Baltimore, Maryland

January, 2008

INFORMATION TO USERS

The quality of this reproduction is dependent upon the quality of the copy submitted. Broken or indistinct print, colored or poor quality illustrations and photographs, print bleed-through, substandard margins, and improper alignment can adversely affect reproduction.

In the unlikely event that the author did not send a complete manuscript and there are missing pages, these will be noted. Also, if unauthorized copyright material had to be removed, a note will indicate the deletion.

ProQuest LLC
789 E. Eisenhower Parkway
PO Box 1346
Ann Arbor, MI 48106-1346

Abstract

This dissertation presents the modeling and development of a test for algorithmic problem-solving skills; an instrument specialized in surveying fundamental abilities in computer programming. This action research study has been driven by an educational need to understand what prevents many college students from learning to elaborate computer programs and has been performed under the premise that a large number of instructional challenges are due to factors indirectly related with coursework like deficiencies in reading, arithmetic, or algebra; abilities which mastery precedes the programming level. The work has been grounded in a constructivist theoretical framework and has followed a hermeneutic approach to understand and integrate common programming errors, programming-specific thinking styles and problem-solving ability domains. The final product, a web-based survey prototype —the initial stage of a screening platform— is already being used to better identify, and start benchmarking, problem-solving skills essential in effective learning of computer programming. The software tool, along with the methodological framework associated, are aimed to provide programming instructors with information and resources to differentiate instruction according to diverse levels of problem-solving abilities, as well as help students to reflect on their problem solving strengths, while gaining a deeper understanding of the knowledge, abilities, and cognitive processes needed to become skillful in creating algorithms and elaborate computer programs.

Advisors: Dr. Susan M. Blunck and Dr. Scott F. Smith

Reader: Dr. Gerald M. Masson

Acknowledgements

This work was supported by the National Council of Science and Technology of Mexico (CONACyT) and the Institute of International Education through the Program Fulbright-García-Robles-CONACyT, scholarship #12345-678; the Encyclopedia Britannica Program for Latin America, and The Johns Hopkins University through the Department of Computer Science, the Whiting School of Engineering, and the Information Security Institute.

This dissertation had not been possible without the invaluable guidance and patience of my advisors Dr. Susan Blunck and Dr. Scott Smith, neither without the prompt intervention of Dr. David Nasrallah, Dr. Elias Shaya, and Dr. Vernon Savage during rough times. The author also wants to express his deepest gratitude to my family, Mario, Luz, and Nora, for their continuous support; to my Professors Dr. Joanne Houlahan, Dr. Stuart "Bill" Leslie, Dr. Gerry Masson, and Dr. Jon Cohen for pointing the way; and to my friends Horacio Jaramillo, Peggy Hayeslip, Stephanie Regenold, Emily LaBathe, Nick Arrindell, Linda Rorke, Debbie DeFord, Cathy Thornton, Steve DeBlasio, and Steve Rifkin, for their continuous encouragement.

This work benefited from lengthy discussions with colleagues and former students, especially Jacobo Hernández, Norma Angélica González, Haydeé Rodríguez, Juan Carlos Valencia, Soraya Assar, Zara Khalid, and Jennifer Bair, but also many others. Thanks to all of them.

Finally, a special recognition to the people who helped me in testing the software and providing valuable feedback about its features and how to improve them: Ingrid Belmont, Steve DeBlasio, Claudia de Leon, Svetlana Goncharova, Jacobo Hernandez, Jaime R. Martinez, Miguel Orozco, Steve Rifkin, Haydee Rodriguez, Juan Carlos Valencia, Nora Vasconcelos, and Jiahui Zhao.

Jorge Vasconcelos

December 2007

Contents

List of Figures

List of Tables

Preface

"When we had a few weak computers, programming became a mild problem, and now we have gigantic computers, programming has become an equally gigantic problem."
Edsger W. Dijkstra, 1972 [174]

As any human endeavor, the development and implications of the computer machine have an inherent cultural value, worthy of teaching and studying in any classroom. However, because of the revolution it has driven, education at literacy level has proven to be insufficient to enable users to successfully interact with information technologies, which support the information infrastructure of today's world. Attentive to this emerging situation, institutions of scholars and professionals have set guidelines to promote thorough education in this field.

The National Research Council (NRC) introduced the notion of *fluency* — deep and dynamic integration of concepts, capabilities and skills to understand and get involved with information technologies [122]— and the Association for Computer Machinery (ACM), in a joint effort with the Institute of Electrical and Electronics Engineers (IEEE), developed a new curricular model for higher education in computer science [1].

However, carrying out the recommendations have resulted in challenges as diverse and profound as the complexities the computers have brought, particularly to what computer programming respects. Hence, despite the best efforts of teachers and schools to respond to this imperious social need, a very large number of students fail to acquire the programming skills expected after introductory courses — understanding a problem description, decomposing it into sub-problems, developing individual solutions and integrating them into a complete one, and then evaluating the whole process and its outcome [76, 87, 97, 110, 117]. International studies of programming skills have reported this as a serious and extended problem, apparently independent of country and educational system [110, 117]; a problem that precludes students from better utilization of computer applications, discourages them from pursuing further studies in computing [76, 87] and, thus, fails to attend industry and academy demands for people versed in information technologies [17, 142].

The recurrence of these situations, both at individual and institutional levels, has led programming instructors to wonder why this activity is so difficult, how to teach it, and how to facilitate its learning.

Throughout 17 years of teaching programming, this author has witnessed a large number of students struggling whenever asked to create algorithms or develop the corresponding programs, being unable to extend, or even to reuse, already working programs, or to explain the reasoning behind them. So, motivated by such experiences, and looking forward to improve his own practice, the author initiated a research study of factors favoring or impeding the process of learning to program

computers, as well as constructivist approaches to address them at the problem-solving level.

Such study involved identifying domains of cognitive abilities related to algorithmic problem solving —the intellectual foundation of computer programming [130]— and designing a screening instrument to identify them during introductory programming courses.

The study, presented to detail in the following chapters, also responds to this screening need, also pointed out by scholars like Mayer [114, p.610], Lister and Fitzgerald [110, p139], Kramer [100, p.42], and Hazzan (commented in [100].) It has expanded our previous constructivist studies in programming education, explored alternative models and techniques for testing problem-solving ability, as well as technical aspects of automated assessment in introductory programming education.

Chapter 1 presents an overview of the current state of computer programming education: the need, the challenges, the main problem and an introduction to the complexities of making programs. The study prompted afterwards is explained in this chapter.

Chapter 2 provides a discussion of the theoretical elements upon which the study is grounded, featuring both, the literature review and the conceptual constructs specific to this study. In addition, this chapter presents a thorough review of the problem-solving issues affecting programming instruction.

Chapter 3 describes the surveying instrument developed to better appreciate problem-solving skills within five ability domains: (*i*) reading comprehension, (*ii*) problem abstraction, (*iii*) algebraic and logic manipulation, (*iv*) stepwise planning,

and (v) process analysis: tracing and debugging. Chapter 4 presents and discusses the results of piloting such instrument.

Finally, Chapter 5 provides further discussion on the results and experience of piloting the surveying instrument and proposes future lines of research.

The conceptual framework developed, along with the methodology followed, constitute a new model of guidelines for computer science educators to reflect on their own practices, evolve their works more efficiently, and gain insights regarding the different levels of problem-solving abilities of their students. In addition, as example of its applicability, a prototype of a computer-based test has been developed, laying the foundation to develop a software platform to screen students' problem-solving skills with relative efficiency and, by periodic application of this surveying system, trace skills development throughout the course.

Chapter 1

Introduction

1.1 The Need for Computer Programming Education

As dependence on information technologies has increased, the level of computer science expertise demanded by society has shifted from passive literacy to active fluency. Nowadays, the workplace demands "knowledge workers," able to solve problems, analyze knowledge, and use technology effectively [158].

According to the study *Being Fluent with Information Technologies,* published by the National Research Council in 1999 [122], besides the need to interact with constantly changing computer environments, specific capabilities in computer programming, algorithmic thinking, and problem solving, are now required to enable users to successfully interact with the information infrastructure of today's world [3, 122, 151, 152]. Furthermore, the Association for Computer Machinery, in a joint effort with the Institute of Electrical and Electronics Engineers, developed a new curricular model for higher education in computer science [1], which specifically points programming skills as particularly fundamental for computer science students (p.22).

Besides being foundational for software development, learning programming promotes intellectual abilities such as general problem solving, methodological reasoning, and logical thinking, essential to understanding and managing the intrinsic complexity of modern information systems, in addition to allowing better utilization of computer applications [151]. Moreover, Knuth [96] has expressed that "this knowledge is preparation for much more than writing good computer programs; it is a general-purpose mental tool that will be a definite aid to the understanding of other subjects" (p.10), Cunningham [42] regards problem solving as "the most important value-added part of computer science education." (p.181), and Kramer [99] claims that the meticulosity acquired by practicing programming and algorithmic thinking, i.e., building and verifying a product step-by-step, can be applied to a wide range of products and services.

1.2 Challenges in Programming Education

As expressed by John Carroll, learning something new is generally difficult, and helping someone else to learn, is even more difficult [34]. Computer programming is one of the best examples of such claim.

The praxis of computer education has shown that instructional difficulties in programming are old, complex, and diverse [6, 16, 27, 34, 72, 150, 177], and found whenever students struggle dramatically to cope with coursework [124, 135], withdraw from courses [81, 120, 177], or finishing them without being able to create any program [30, 94, 117, 109].

1.2.1 The Classroom Battleground

The literature in this field is profuse with examples and anecdotes reflecting the hardships of learning to program computers. For example, Palakal [124] points out that students' attention goes to learn either theoretical concepts or programming language (p.1), Proulx [135] comments about the frustrating experience of meeting bright students who become lost when asked to write even the simplest program (p.80), and Barnes [10] cites the common case of students who say "I just don't know how to start with this task..."

Furthermore, Buck [29] and Linn [108] refer the multiple times students just do random attempts to produce a working program (p.17 and p.121, respectively), Thomas [161] and Jenkins [90] mention how many students finish courses vowing that their final projects or future careers will not include programming at all (p.53 and p.3, respectively), and Lister [109] states that "stronger [programming] students are not challenged while the weaker students flounder." (p.143).[1]

Situations like these have become increasingly evident as academic institutions keep incorporating computer science and information technology courses into their curricula, and have made scholars around the world wonder why this activity is so difficult, how to teach it, or how to facilitate its learning.

1.2.2 Unclear Expectations

Oftentimes, programming goals have been set according to instructors' preferences or experience [28, p.75; 164], programming tools availability or industry trends, and,

[1] Each of the issues previously mentioned has been repeatedly experienced and commented by the author and colleagues [76, 87].

consequently, computer science literature became very diverse regarding learning objectives. In the cases of problem solving and programming skills, they have often treated in generic terms without clearly defining them.

In 2001, the *ACM-IEEE Computing Curricula 2001 for Computer Science* [1] set a general outline for courses on programming fundamentals, which included topics on algorithmic problem-solving process and programming constructs. The document also stated cognitive capabilities and skills to be expected from computer science graduates: knowledge and understanding, modeling, requirements, critical evaluation and testing, methods and tools, and professional responsibility.

This very same year, Bailey and Stefaniak's [9] identified 85 concrete skills that industry regarded as important assets for programmers. The top five were: (*i*) ability to read, understand, and modify programs written by others, (*ii*) ability to code programs, (*iii*) ability to debug software, (*iv*) listening skills, and (*v*) problem-solving process (specifically, identify and analyze problems, and design of decision trees.)

However, expectations have not become clear yet. In a reflection about current practices in software development, Clear [38] even wonders "what is programming?" while Thompson, Hunt and Kinshuk [164] present different conceptions of learning this activity, such as solving problems, learning a programming language, or becoming part of a technical community.

Recently, Stone and Madigan [158] noticed that, despite workplace demands "knowledge workers" (i.e., people able to solve problems, analyze knowledge, and use technology effectively) there are disturbing inconsistencies among state standards, student abilities, and student perception of their information technology abilities. It

is worthy of mention that their study focused only in elementary tasks (like saving documents, using antivirus, or performing internet searches) without even entering in the realm of problem solving or programming.

1.3 The McCracken Study

The most dramatic instance of the instructional issues in programming was acknowledged, and widely exposed, by a working group of the 6th Conference on Innovation and Technology in Computer Science Education (ITiCSE 2001), that performed a study on whether programming students could actually elaborate programs [117].

The Multi-national, Multi-institutional Study of Assessment of Programming Skills of First-Year CS Students started by setting up a framework of learning objectives for introductory programming courses (fig. 1.1), according to the ACM-IEEE *Computing Curricula 2001* [1], and then testing programming ability through a trial assessment administered to college students in several universities in different countries. Lack of positive results led the group to express "many students do not know how to program at the conclusion of their introductory courses... This implies that it was the students' knowledge, rather than their skills, that enabled them to successfully complete their first year courses." (p.125 and p.134.)[2] Subsequent analysis revealed important clues on what seemingly prevented students from completing the requested programs: (*i*) difficulties in abstracting the

[2] The McCracken group is referring here to factual knowledge (the *know-what*) only as knowledge and to procedural knowledge (the *know-how*) as skills [70]. Ala-Mutka refers to them as programming knowledge and programming strategies, respectively [5].

problem out of the exercise statement, (*ii*) inability to implement the planned strategy, (*iii*) insufficient time to solve the exercise, and (*iv*) unsuccessful interaction with the programming environment.

1. **Abstract the problem from its description**: Identify relevant aspects from the statement and model them in the proper abstraction framework.
2. **Generate sub-problems**: Break the problem down into simpler, more manageable ones.
3. **Transform sub-problems into sub-solutions**: Design detailed strategies to solve each sub-problem and implement them with a computer language.
4. **Re-compose the sub-solutions into a working program**: Integrate (correctly) every piece of the solution into one program.
5. **Evaluate and iterate**: Test and debug the program until it works correctly, then determine if the whole process has led to a good solution.

FIGURE 1.1. McCraken's Model of the Problem-Solving Process in the Domain of Computer Science [117]. Analogous models appear in [41, 30, 54, 111].

1.4 On the Complexities of Making Programs

1.4.1 Overview of the Activity

Programming a computer is much more than just assembling instructions for enabling a machine to perform a particular job. In its most general form, programming encompasses four fundamental aspects: (*i*) understanding the job to be accomplished, (*ii*) specifying a detailed plan that can be carried out by the computer, (*iii*) mapping the plan into the constructs of a programming language, and (*iv*) using a programming environment to transfer the mapped plan to the computer and verify if the goals have been fulfilled [149].

1. In order to understand the objective of the desired program, the programmer must become acquainted with the fundamentals of the discipline related to the problem, and comprehend what exactly is being requested.
2. To make a plan, the programmer must have, at least, minimal knowledge about what the computer is able to do, and the ability to properly put those operations together to reach the desired goal.
3. To implement the plan, the programmer must know the words and rules of a programming language, and should know how they represent computer operations and their effects on the data.
4. Both plan and program are just theoretical entities, unable to perform anything until properly transferred into a computer, which is achieved by means of software tools. Thus, the programmer requires some dexterity in using these tools. Finally, to review the program, the programmer must keep in mind the global goal; identify the origin and nature of errors, and predict outcomes.

1.4.2 Multiplicity of Knowledge Domains

The cognitive demands of each aspect of programming are diverse and can be classified within four different domains of knowledge: (*i*) methodological problem solving, (*ii*) programming language, (*iii*) programming environment functionality, and (*iv*) conceptual background specific to problem. Such distinction is necessary to better understand the challenges faced by programming students (see table 1.1.)

To succeed, the student needs to become skillful in each domain, i.e. learn, understand, and effectively apply concepts and techniques from each one, usually simultaneously and in a short period of time. Not being difficult enough, issues in any domain can hide student's particular strengths and weaknesses in other domains, leaving instructors unable to distinguish where the student actually needs help.

Domain	Concepts/Skills to be Developed
I **Methodological problem solving**	• Become skillful at creating problem-solving strategies. • Become aware of the mental processes required to make such creations. • Being able to visualize the sequence of actions a strategy will require.
II **Programming language**	• Learn syntax and semantics of a formal language. • Develop ability to find and apply additional programming resources.
III **Programming environment functionality**	• Learn the to interact with the development platform. • Become skillful at reading and interpreting error messages.
IV **Problem-specific conceptual background.**	• Develop the ability to gather information not specifically stated like the context of the problem or the theory behind it.

TABLE 1.1. Knowledge Domains for Introductory Programming.[3] Some aspects of this model also appear in [5, 27, 105, 149, 178].

[3] In a recently study, Gray, et al [77] have briefly analyzed different aspects of programming, which they called "dimensions."

Because of this situation, the challenges in programming education should be analyzed from four different perspectives: as a problem-solving skill, as a phenomenon of (formal) language acquisition, as a result of computer interaction, and even by its psychological factors (like attitude, predisposition, fear, anxiety, etc.)

1.5 Factors Affecting Problem-Solving Ability

1.5.1 Problem Solving and Programming

Solving problems is a pervading and recurrent aspect of programming, which requires ability to integrate and apply a number of fundamental concepts, cognitive skills and thinking styles. Developing abilities to solve problems in methodological fashion is crucial to success in making programs and scholars have been concerned with this issue for long time [10, 31, 42, 116, 132, 140, 163, 166]. In general, problem-solving strategies seek to make explicit the intellectual stages involved in transforming givens into outcomes: understanding the problem, devising a suitable plan, executing it, and then reviewing the outcome along with the plan (fig. 1.2).

The magnitude of the complexity to create programs can be better appreciated by regarding the programming lifecycle and a problem-solving strategy altogether (fig. 1.3).

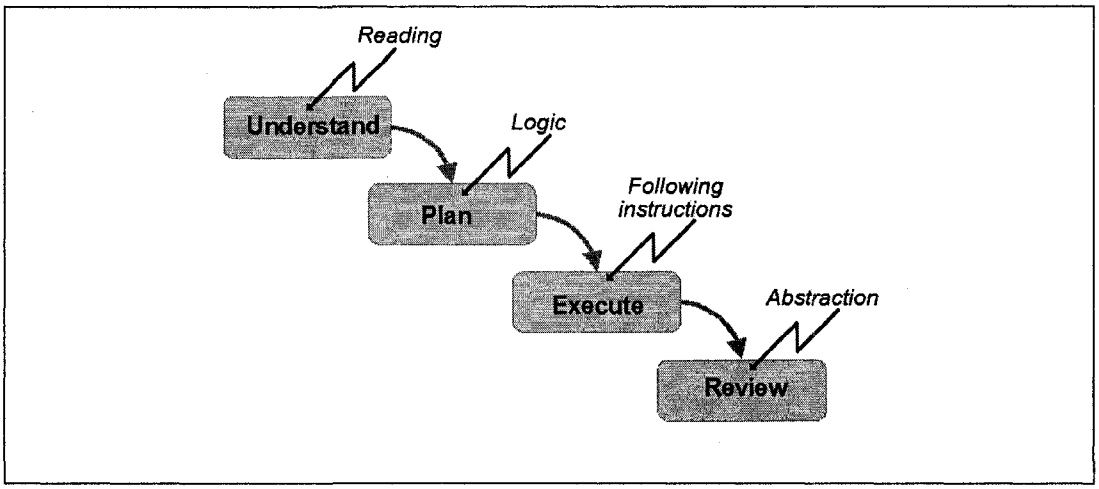

FIGURE 1.2. Polya's Classic Strategy to Solve Problems. Solving problem does not tend to be a simple and self-evident process. George Polya developed a strategy to guide students' thinking and facilitate resolution of mathematical problems [132].

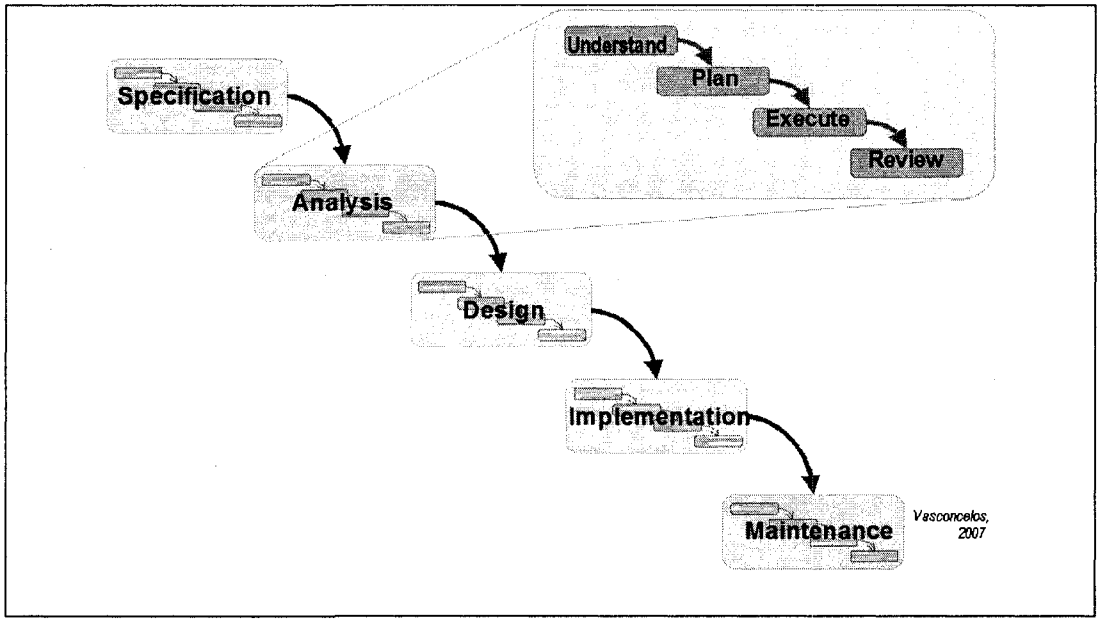

FIGURE 1.3. Problem-Solving Throughout the Programming Lifecycle. Good programs result from working through several phases. Similarly, solving problems methodologically implies following a sequence of stages. However, when tasks involved are non-obvious, each becomes another problem to be solved. Therefore, making a program to solve one problem may actually require solving twenty more.

1.5.2 Fragility in Problem-Solving Skills

The development of skills to solve problems can be deterred by a number of factors, such as misconceptions —knowledge or ideas that counteract the actual problem solving process— or fragile knowledge —concepts the person knows but fails to apply effectively.

According to Stone [158], many students start college with a level of information technology skills insufficient for academic success in computer science courses (p.76). Furthermore, many students undergo programming courses with deficiencies in elementary abilities to solve problems like reading, arithmetic or logic [41, 30], consequently increasing the complexity of the course (fig.1.4.).

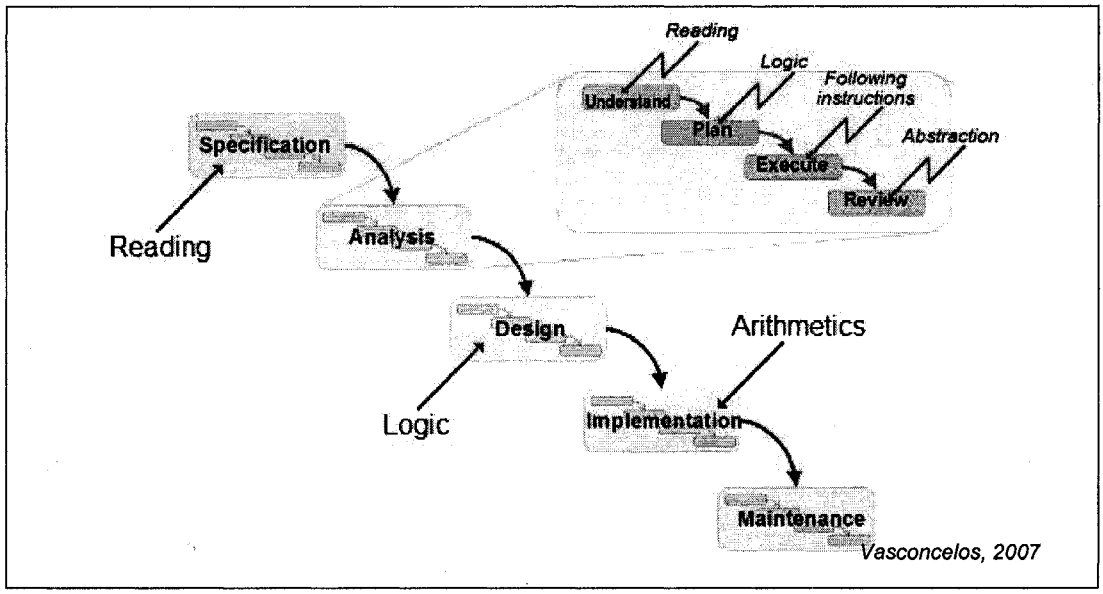

FIGURE 1.4 Factors Affecting Problem-Solving and Programming. The figure illustrates how reading difficulties can prevent understanding the problem during the specification phase, flawed logic impacts the design of the algorithm, and arithmetic errors (like inconsistent use of operators hierarchy) affect the implementation. It also shows how no plan can be prepared if the problem is not completely understood first, and similarly, no good review occurs without an understanding of the whole process.

Several educators have described their classroom experiences related to this situation. For example, Burton [30] pointed out that some of the basic mathematical skills needed to create programs are not effectively applied (p.113), among them familiarity with symbols, variables, relationships, and functional notation. The same scholar mentioned how an inadequate command of English can cause misinterpretation of compiling and debugging messages (p.112). Krishna [101] found severe difficulties in the understanding of operator precedence and associativity, both mathematical concepts fundamental to create correct computational expressions. Lane [106] observed how many students engage in problem-solving activities without first having a global picture of the task to be accomplished. Winslow [178] referred to the inability of some students to express a plan in a step-by-step, program-like form (p.17), and Pea's description of intentionality errors [127] can be interpreted as flaws in logical thinking (i.e., instruction outcome is not directly inferred from the preceding instructions).

In addition, inability to create programs due to fragility to read and perform systematic analysis was confirmed by a working group of ITiCSE 2004 [110]. This group performed a study involving college students in several universities in different countries. Their report recommended the inclusion of special screening mechanisms in any research project studying problem-solving skills in the context of computer science education.

1.6 Study Description

1.6.1 Problem of Concern

In brief, the observation motivating this study is that many programming students struggle because of problem-solving factors whose mastery precedes the programming level, such reading skills, arithmetic and algebraic abilities, or logical thinking.

The study has evolved through two main phases. For the first phase, we focused on finding out what concepts and skills students have at the beginning of such courses, and then we observed their impact in the instruction process in a constructivist way. We believe that having first some understanding about those cognitive pre-requisites for computer programming is necessary to address the problems arising during the courses.

For the second phase, we explored alternative models for surveying the problem-solving skills required in programming, aiming to detect strengths and weaknesses, as well as technical aspects of computerized classification testing.

It is worthy of mention that, although it is not clearly stated in the literature, the notion of surveying tends to imply gathering data or qualitative information, while assessment tends to refer quantitative evaluation of knowledge or dexterity. Thus, because the qualitative nature our study, the term "survey" has been preferred over "evaluation" or "assessment."

1.6.2 Initial Research Question

The main research question behind the study has been *what prevents students from being ready for programming courses?* However, because of cognitive demands of the field (see section 1.4), there are too many inquiries to be answered, for example, *what prevents students from* (*a*) *understanding computer-programming concepts?* (*b*) *acquiring computer-programming skills?* or (*c*) *successfully interacting with computer-programming environments?* A comprehensive approach to the main research question would require thorough studies on all these three aspects at several educational levels. However, the scope of our work was limited to the problem-solving aspects of acquiring programming skills: *what elementary problem-solving factors prevent college students from acquiring computer-programming skills?*

We have approached this problem from several angles: (*i*) identification of independent cognitive elements requiered by computer programming, (*ii*) definition of a set of capabilities indispensable for successfuly solve problems, (*iii*) regard the waterfall model of program construction as a sequence of problems, each in need to be solved to finally have a sound program and (*iv*) identify recurrent issues within the programming lifecycle, and develop a set of variables that may indicate posibility of failure in specific parts of the sequence.

1.6.3. The Exploratory Study

The objectives of the first part of the study had been, first, to identify pre-programming knowledge and problem-solving skills that can facilitate or challenge

the instruction process, and second, to develop a reliable and systematic mechanism to recognize them within students' coursework.

To this end, a questionnaire was designed to survey five ability domains involved in problem solving: (*i*) reading comprehension, (*ii*) problem identification, (*iii*) algebraic manipulation, (*iv*) stepwise planning, and (*v*) process analysis: tracing and debugging. Rather than assessing knowledge or dexterity, the questionnaire was intended to find error patterns and trends that could indicate skill fragility or potential learning hazards. The analysis performed after the pilot application served to set an inventory of common pitfalls and propose a taxonomy of cognitive skills and thinking styles related to programming —intellectual abilities allowing students to get involved with the programming activity and, consequently, with the instruction process. The questionnaire elaborated served as foundation to perform the second part of the study (see Appendix A for details).

1.6.4. The Final Study

To expedite the surveying process, computer-based testing techniques were explored, which led to a secondary research question: *What problem-solving skills can be effectively identified through objective testing?* We explored three venues within this phase: (*i*) educational aspects of introductory programming, (*ii*) identification of student's ability to solve programming-like problems, (*iii*) and the application of computer based surveying techniques within a constructivist assessment framework.

Chapter 2

Conceptual Framework

2.1 Introduction

This chapter provides a discussion of the theoretical elements upon which the study is grounded, featuring both the literature review and the conceptual constructs specific to this study.

Research methods and processes of the study where shaped through action research and reflective practices, and were grounded in a conceptual and methodological framework, consisting of: (*i*) an educational philosophy to guide the study —a sound bond between constructivist epistemology and computer programming— (*ii*) a new model of educational elements involved in introductory programming, (*iii*) an observational tool to perform the study —an instrument specialized in surveying algorithmic problem solving skills— and (*iv*) specific variables to observe while the study was being developed —inventories of common programming pitfalls, cognitive and practical skills, and thinking styles specifically related to programming.

2.2 Research Tools to Study a Dynamic Topic

Because of the dynamic nature of the study, it required the use of research approaches designed to evaluate computer science education: action research and reflective practices to guide understanding through a hermeneutic cycle, adjusting methods, processes, and premises, as new results were providing a better understanding of study's object.

2.2.1 Hermeneutics: Search for Understanding

According to the hermeneutic paradigm [7, 44], the understanding of phenomena follows a cyclic process that starts with preliminary ideas and assumptions, which are verified, enriched or confronted, by means of practical exploration and literature reviews (fig. 2.1). In turn, this leads to better insights about the phenomena, enabling the formulation and testing of new assumptions. Thus, with each iteration of the cycle, our understanding of the objects evolves, allowing us to adjust methods and specific processes as needed.

2.2.2 Action Research: Observing while Practicing

Because the study had sought answers to educational problems while the process was actually taking place, an action research approach was employed. This approach seeks intentional learning from experience, and promotes exploration and testing of new ideas, methods, or materials, as well as immediate assessment of their impact to act accordingly.[4]

---

[4]Action research was introduced by social psychologist Kurt Z. Lewin in the paper "Action Research and Minority Problems" (*Journal of Social Sciences*, 1946), describing it as "a comparative research on

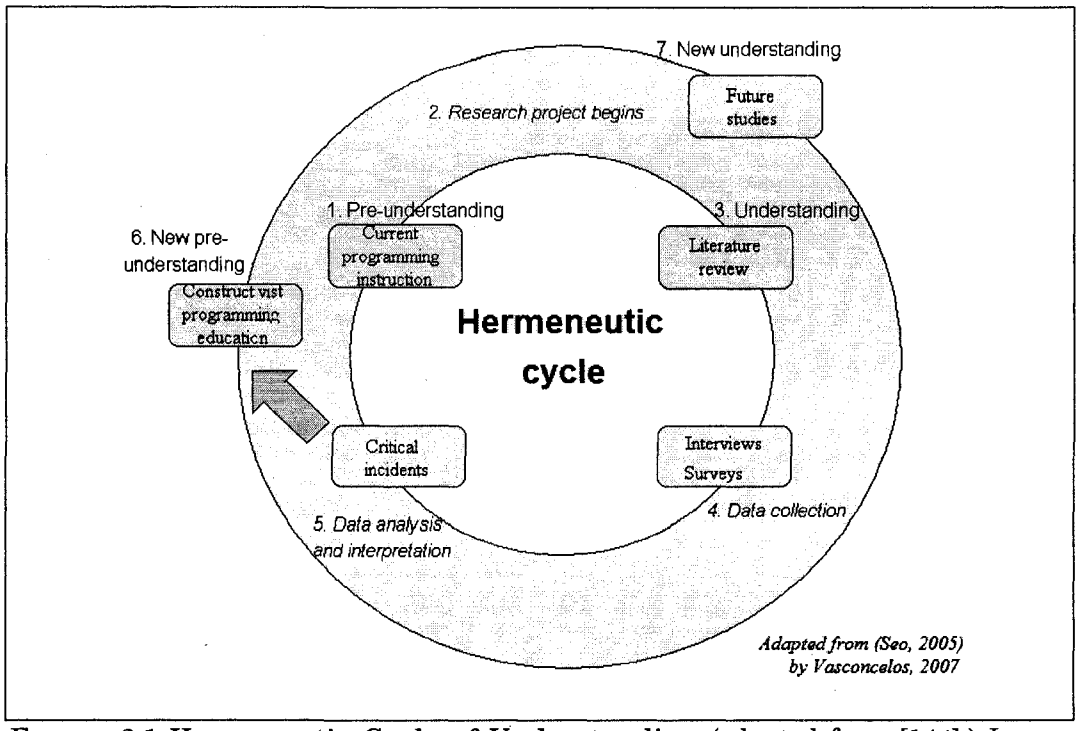

FIGURE 2.1 Hermeneutic Cycle of Understanding (adapted from [144].) In our study, preliminary assumptions were that programming instruction is affected by lack of exposure to algorithmic problem solving and individual differences. Critical incidents occurred after thorough review of the constructivist paradigm and surveying problem-solving skills, which in turn leads to appreciate how basic problem-solving skills impact the programming activity at both practical and cognitive levels.

Action research practices transform the researcher in a "participant-observer" who studies educational phenomena through the cycle: (*i*) observation (data collection), (*ii*) critical reflection (evaluation), and (*iii*) action [71, 159]. Hence, the research study can be designed in terms of actions to be taken, to later collect, analyze and interpret data resulting from such actions, and then drawing the corresponding conclusions [83].

---

the conditions and effects of various forms of social action, and research leading to social action." (Online source: http://www.infed.org/thinkers/et-lewin.htm, last accessed: 10/9/2006.)

2.3 The Constructivist Paradigm

Constructivism is a philosophical system addressing the question "how do we know?" According to the paradigm, knowledge is constructed rather than imprinted [143, 170, 171], which occurs through a mental process that associates new concepts or ideas with existing ones [64]. Thus, learning is an individual experience, in which knowledge is built in a recursive way, i.e. new facts, ideas, beliefs, or skills, are built up from previous ones [19, 20]. The construction can be conscious, by reflecting on our experiences, or subconscious, by getting used to some situations.

From an epistemological point of view, according to the constructivist paradigm, no knowable object can be regarded as "truly real" [5] because, in order to learn about it, the object has to be transformed into a mental entity able to be handled by our psyche (and each mental model differs from one person to another.) As long as the phenomena, explanations or effects, are consistent with our current model (i.e., they are seen as "natural" to us), we can understand and learn from them. By the same token, our interaction with the world does not occur directly[6], but through models in our brain that allow us to understand the situation and act accordingly (namely, the phenomenon "makes sense".)

Models are not duplicates of reality, just ways of explaining and interacting with it. As long as they serve their purpose, they are regarded as viable. However, if a cognitive conflict occurs due to new or unexpected situations, the model is

---

[5] From a constructivist standpoint, ontological reality is either irrelevant or rejected [20].

[6] In fact, interaction with the world cannot occur in direct way. Because of nature, perception occurs through a sensory layer that buffers the interpretation and control centers. This phenomenon transforms *the actual world* into a *perceived world.*

considered non-viable and requires constructing a new one (if dealing with the new situation is desired.)

In this way, learning is the process for which mental models are enriched or created, while teaching is the process that helps or promotes learning. The success of these processes is contentious to make sense of new facts, or ideas, and incorporate them into the new model. Thus, because each person has different preliminary models and association processes, creating a model is an individual experience, and therefore, that learning actually occurs in different ways with each person.[7]

According to this perspective, teaching should be adapted to regard different mental models. Consequently, the teacher is no longer an information source but a facilitator or motivator [91], someone who promotes (or coaches) the process of building knowledge. In order to accomplish this, the teacher has to acknowledge that learning is indeed an individual experience, be concerned about the way each student learns, and also identify preliminary knowledge the student poses to prevent new information from colliding with students' mental models, reinforcing misconceptions, or a failing learning process.

2.4 Educational Elements for Programming

A thorough analysis of available literature, research trends and teaching experiences led us to develop an operational model of educational elements involved in introductory programming (fig. 2.2). This model served two purposes: first,

[7] In consequence, "learning to program is a unique experience for each student, and is not fully understood why one person in an introductory programming course learns to program better and more quickly than the next." Ramaligan, V., et al. "Self-Efficacy and Mental Models in Learning to Program." *ITiCSE'04*. p.171.

organize, clarify and set the boundaries of the study, and second, facilitate the identification of elementary problem-solving abilities that have major impact in the programming activity or its learning.

Our model is dividing programming education after Wirth's classic work [179]: (*i*) instruction in algorithms, and (*ii*) instruction in data structures. Currently, the model only emphasizes the algorithmic branch because, besides being foundation of computer science and information technologies [22, 130], the thorough development of algorithms and the application of its characteristics constitutes a sound methodological framework known as *algorithmic thinking* [43, 95, 99, 169], from which we elaborated only the procedural aspects.

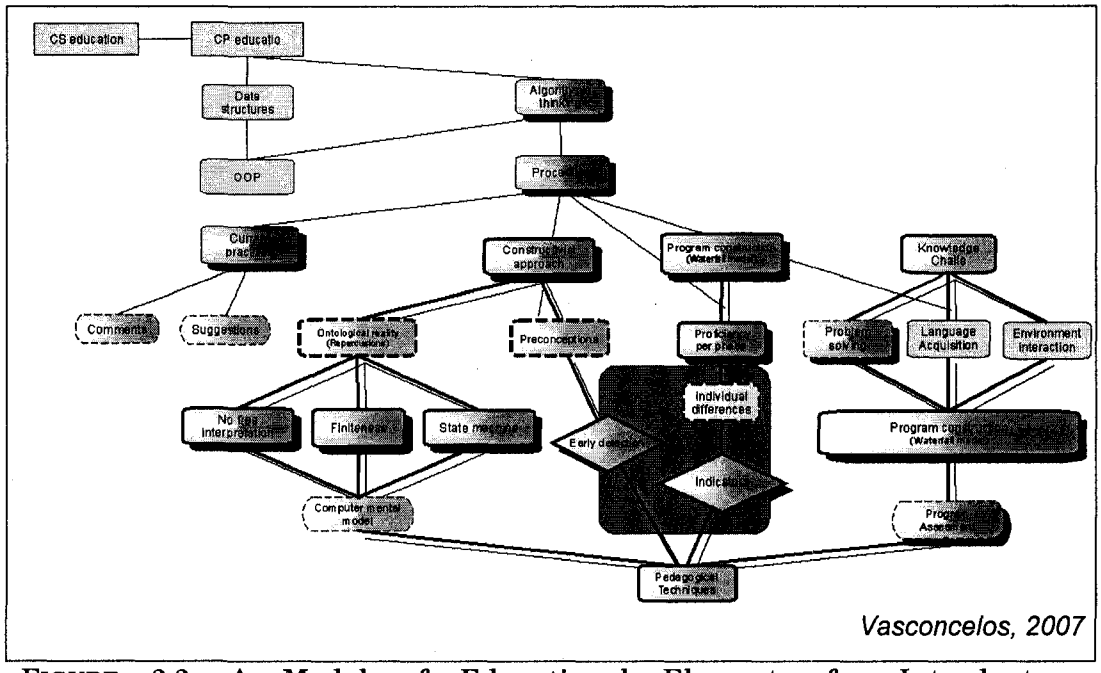

FIGURE 2.2. A Model of Educational Elements for Introductory Programming. The model summarizes the results of the first five stages of the hermeneutic cycle illustrated in fig. 2.1, applied to our study.

The model identifies three factors as necessary for a thorough study of the programming activity or its teaching: *(i)* theoretical foundations of education, *(ii)* educational aspects specific to program construction, and *(iii)* knowledge domains required by programming. Within the theoretical foundations, an educational paradigm is needed to shape both, the research study and the teaching practice. Further more, it helps to understand how learning programming is affected by prior knowledge, personal experiences, and also by characteristics specific to the computer machine (see section 2.3). In the other end appear three knowledge domains that programming students have to learn simultaneously: algorithmic problem solving, formal language acquisition and interaction with programming environments [5, 27, 178].

With respect to program construction specifics, the review of a lifecycle model served to isolate critical points, i.e., specific tasks of the programming activity where fragile problem-solving skills can become learning hazards (see section 2.5). As explained in the first chapter, each stage in the lifecycle is in fact a problem the student has to solve until the actual program is finished. However, because of the sequential nature of the model, issues at any particular point are carried on through all subsequent stages, making difficult, even impossible, to create the program.

The model narrows down the boundaries and scope of our study. Figure 2.3 illustrates the point where problem-solving skills and computing pre-conceptions should be screened in order to differentiate pedagogical techniques properly [178].

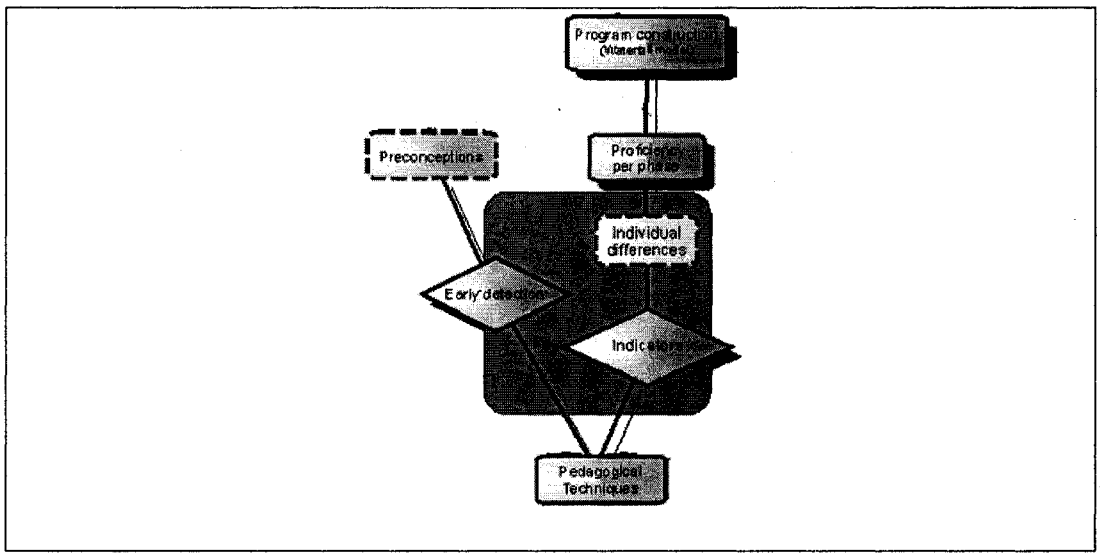

FIGURE 2.3. Screening Development of Problem-Solving Abilities to Differentiate Programming Instruction.

2.5 The "Waterfall" Lifecycle Revisited

The disciplines of systems engineering and software engineering have shown that (sound) computer programs result from working through several phases known as lifecycle models [73, 134, 160]. Such models help organize complexity when elaborating products that must meet multiple requirements, as well as monitor effectiveness of the different tasks involved in such process. In our study, a programming lifecycle helped to analyze this activity from a problem-solving perspective and identify critical points where common learning hazards occur. For simplicity, we revisited the traditional waterfall model of program construction[8] [24, 134, 167), however, the analysis also applies to the spiral model of programming [134].

[8]Although the waterfall model now belongs to the common knowledge of the computing field, it tends to present modifications depending on particular textbooks or instructors. Some interesting variations appear in [63, 165]. A discussion regarding its current applicability appears in [38].

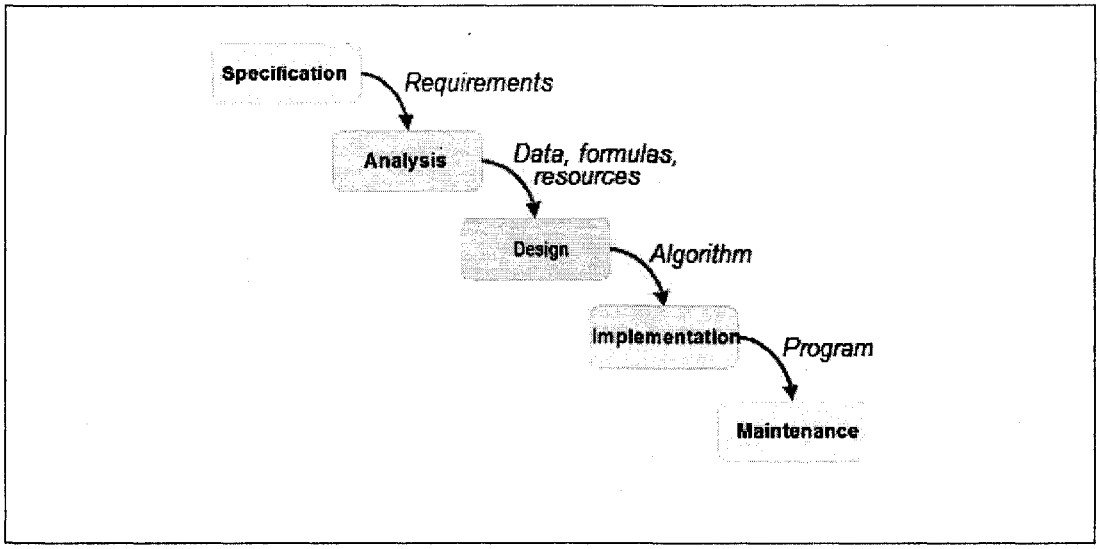

FIGURE 2.4. The Waterfall Model for the Lifecycle of Programming. The simplest approach to elaborate programs is the "waterfall model" [167]. Programming is depicted a sequence of "black boxes" where the output of each box becomes the input of the next one.

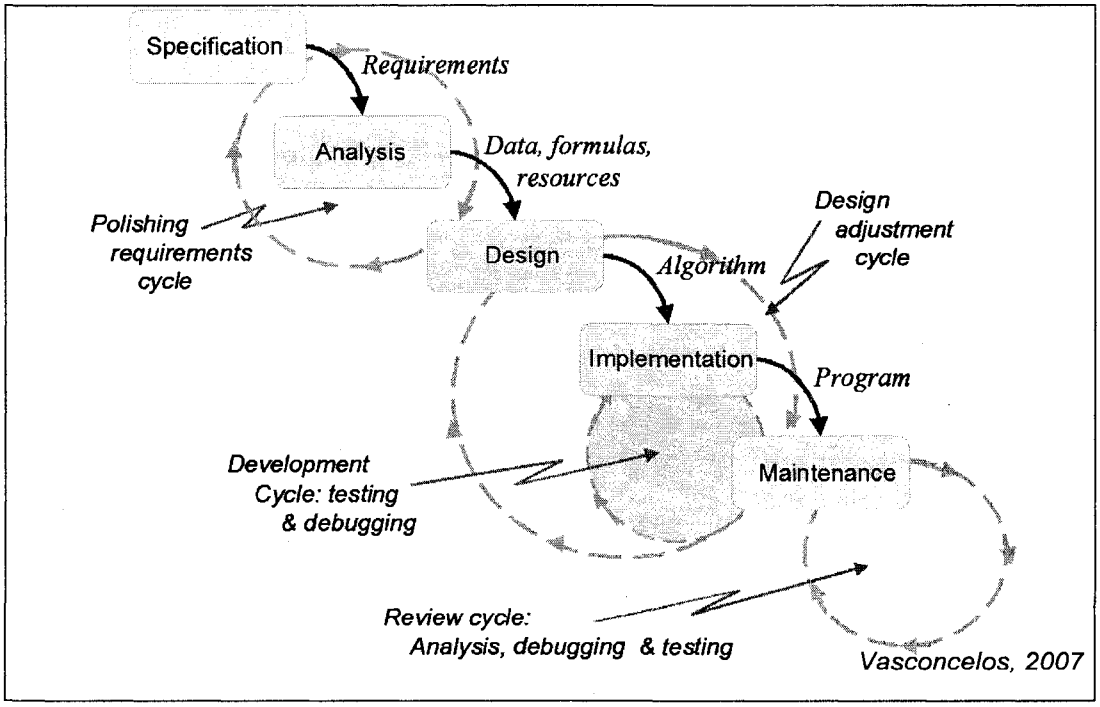

FIGURE 2.5. The Waterfall Model of Programming with Refinement Cycles. Depending on the complexity of the requirements, the "waterfall model" can become iterated. Some students become stuck in the cycles, without further progress in the program under construction.

This classic model (fig. 2.4) describes the making of a program as a result of following a sequence of stages, each producing a prescriptive document to guide activities in the next phase [2, 18, 24, 46, 58, 167]. In principle, success at any given point is dependent upon (successful) completion of the previous stages; consequently, succeeding in all phases of the lifecycle leads to a working program (if not an efficient one, at least one that meets the requirements).

The waterfall model works well to elaborate programs with few requirements. However, as their number increases, elaborating the program demands the programmer to move back and forth in between stages (fig. 2.5).

Because the tasks involved in each stage are not self-evident, every stage becomes a problem by itself. Therefore, besides the one for which the program was required, the student has to solve, at least, six more problems. Due to the sequential nature of the model, any issue occurring at some particular point is carried on throughout subsequent stages,[9] a "snowball effect" with serious repercussions for programming education: fragile problem-solving skills increase the complexity inherent to each stage of the lifecycle [106], which can prevent the completion of the program, understanding how to make it, or how it works.

[9]Teaching experiences have shown that recovery is not always impossible, however process tends to be difficult and unclear.

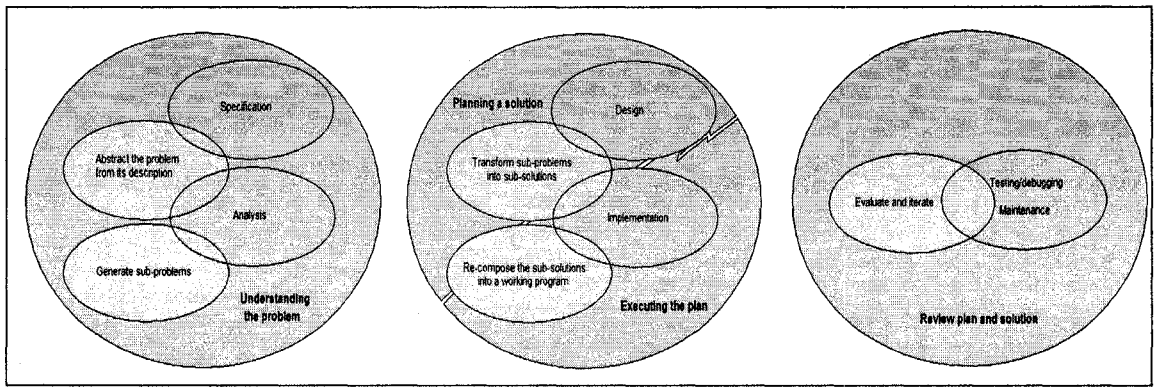

FIGURE 2.6. Relationship among the McCracken and Polya Models, and the Stages of a Programming Lifecycle.

Figure 2.6 uses Venn-diagrams to illustrate the relationship among Polya's problem-solving strategy, the problem-solving process described in the McCracken study (section 1.3), and the stages of the programming lifecycle[10]. Thorough analysis of teaching and research experiences regarding these models serve to pinpoint recurrent issues challenging introductory programming instruction (see table 2.1).

For example, difficulties in abstracting a problem out of the description, as noticed in the McCracken study [117], can be traced to language deficiencies [27] or plain "shyness to ask questions" [10, 12]; while the inability to elaborate a program that implements a sound algorithm can be due to a seemingly obscure programming language [26, 125], to an "unfriendly" programming environment [67, 126, 181], or to issues reading and interpreting error messages issued by the compiler [27, 104].

[10] A study performed by Deek [50, 51] presents an analysis of a common model of problem solving and the tasks of program development. Because of their similarities, the study merges them as a "dual common model for problem solving and program development."

Phase	Challenges
1 **Specification** (Stating the problem)	1. Issues reading, or understanding, the problem statement [27, 41, 106, 178]. 2. Difficulty to extract or summarize information from context [72, 117]. 3. Inability to request either clarification or more information ("shyness to ask questions") [11, 12].
2 **Analysis** (Reviewing problem's context)	1. Inability to request clarification or get more information. 2. Inability to break a problem down into simpler units [41, 75]. 3. Lack of knowledge, or intuition, regarding theory involved in the problem to be solved [136, 178]. 4. Difficulties to discern between useful and not useful information [178].
3 **Design** (Planning an algorithm)	1. Problems devising a suitable strategy [41, 106, 117, 153]. 2. Difficulty to express thoughts [76]. 3. Confusion with algorithm characteristics. 4. Assumption that computing agent has preliminary knowledge of the solution process [127, 155].
4 **Implementation** (Programming or *pseudocoding*)	1. Difficulties grasping the computer language [26, 121, 125, 104, 155]. 2. Difficulty to express a plan using a computer language. [117, 155, 178]. 3. Unsuccessful interaction with the compiler or the programming environment [67, 89, 126, 181]. 4. Misconceptions on computer language semantics [13, 78].

TABLE 2.1. Common Challenges to Elaborate Programs.

Phase	Challenges
4 Implementation (Testing or debugging)	1. Lack of intuition, or ability, to predict results from a specific algorithm [8, 80]. 2. Lack of ability, or interest, to verify the logic behind an algorithm and its results [40]. 3. Missing common errors (initialization, arithmetic expressions, loops, etc.) and checkpoints [20]. 4. Inability to hand-trace [66, 110] or difficulty to follow step-by-step procedures [36].
5 Maintenance (Updating, improving, debugging)	1. Issues reading programs or understanding work done by other people [52, 61, 103, 110, 138, 178]. 2. Difficulty to extract, or summarize, information from an algorithm [106, 110]. 3. Inability to request clarification or more information. 4. Lack of knowledge, or intuition, regarding theory involved [178].
6 Documentation (Preparing material for future understanding of the program)	1. Issues writing or understanding the writing process [14, 61, 110]. 2. Difficulties to extract, summarize, or explain information from work that has already been done [52, 110]. 3. Inability to request clarification regarding documentation style. 4. Problems making document outlines. 5. Difficulties to express thoughts in written form. 6. Misconceptions regarding English, computer language or algorithmic language. 7. Lack of intuition or ability to explain what it is expected from an algorithm [110]. 8. Lack of ability, or interest, to explain the logic behind the algorithm (i.e., how the program works) [153].

TABLE 2.1. Common Problems to Elaborate Programs (cont.)

2.6 Research in Computer Science Education

Broadly speaking, scholars involved in computer programming education are trying to answer one of the following questions: (*i*) How to teach computer programming? (*ii*) How to facilitate instruction in computer programming? and (*iii*) Why programming is difficult to teach and learn? Hence, common research goals can be perceived, such as the identification of technical and cognitive abilities to succeed in programming courses, how to help students to better understand the way information technologies work, and provide teachers with tools to recognize and facilitate development of students' conceptions and skills.

2.6.1 Overview

The attempts to facilitate learning to program have moved from creating new languages to designing a diversity of programming environments [56]: Kemeny and Kurtz' BASIC (1964), Papert's Logo (1967), Wirth's Pascal (1970), Pattis' Karel the Robot (1981), Pausch's Alice (1995).

Also, a number of studies have been performed on the psychology of computer programming [112, 113, 114, 115, 146, 153, 173]. The impact of those works seems to have been limited, in part because they are not well known, or were done more than two decades ago. Nevertheless, many conclusions are still valid and can shed some light to understand current experiences, or serve as starting point to analyze current practices.

Contemporary research tends to fall within five main groups: computer interaction [32, 33, 56, 79, 126], computer languages [78, 125], intelligent tutoring [57, 106, 131, 180], lab-augmented class [27], and computing mental models

[13, 21, 150]. Recently, special education has also started to be studied [60]. Several large-scale studies, spanning several institutions, have started to emerge [65, 110, 117], as well as some comprehensive literature compilations [65, 79, 93].

2.6.2. Programming and Problem Solving

Scholars have attributed the challenges in programming education to a number of causes, ranging from insufficient exposure to algorithmic problem-solving, to semantic conflicts with programming languages, and even motivational issues. The studies performed by Mayer [115], Soloway [153], and Ala-Mutka [5] have done comprehensive reviews on several problems of learning and teaching programming, while Winslow [178] presents a psychological overview of programming, containing insightful aspects regarding the pedagogy of problem solving. Table 2.2 sumarizes several programming issues related to problem-solving and their possible origins.

2.7 Problem Solving

2.7.1 Capabilities for programming

Like many scholar courses, programming ones seek to hone a number of cognitive abilities into particular skills. Those abilities need to be present by the time the course starts (they are hardly acquired or developed during the course), otherwise objectives are hardly met and students are likely to fail.

Problem	Possible origin
Poor problem-solving skills.	Lack of exposure to algorithmic thinking [30, 41].
	Early introduction to programming language [115].
	Inability to start/overwhelmed by problem [6, 10, 86].
	Inability to plan [5, p.4].
	Inability to integrate individual steps/instructions [78].
	Inability to decompose a whole into constituent parts [75].
	Inability to follow step-by-step procedures [34, 36].
	Insufficient time [30, 117].
	Defficient pre-requisites (mathematics, English, information technology, etc.) [30,158].
	Errors due to misconceptions [127].
Misconceptions about programming and computing.	Concepts not matching techniques [5, p.4].
	Alternative concepts for correctness [97].
	Inadequate computer mental model [13, 19, 20, 21].
	Learning curve to match program with the mechanism it [59].
	Defficiencies in computer literacy [30].
	Semantic or syntax [4, 6, 40, 67, 78, 115, 119, 125, 135, 153, 155].
Conflicts with programming languages.	Semantic differences with respect to ordinary speech [5, p.3; 19, p.51; 23; 26].
	Language complicates algorithmic solution
	Programming paradigm [5, 30].
	Programming systems not designed for usability [56, 73, 125, 181].
Conflicts with programming tools.	Confusing syntax errors [104].
	Adversity [82, 118, 133].
Psychological or socio-cultural factors.	Overconfidence [5, p.2].
	Student's behavior [128].
	Motivational issues [6, 45, 90, 91].

TABLE 2.2. Summary of Factors Affecting Programming Education.

In the case of introduction to programming, we have been distinguished five big groups, or domains, of problem-solving capabilities (fig.2.7):

1. **Problem comprehension**: During the course, the student needs to read and understand many problem statements (also process descriptions), and get essential information out of the problem context. Related to this

ability, is the one to write down information gotten from that context. A discussion on the importance of reading and related issues appears in [14, 30, 52, 103, 141]. A discussion on issues about problem identification appears in [10, 12, 25, 107, 108, 137, 141].

2. **Functional decomposition**: Regardless of programming paradigm, the student requires the ability to see "the big picture" that any problem represents, and methodologically decompose it into sub-problems. Then, figure out a plan of attack for each one. Some references to this ability appear in [41, 75, 117].
3. **Numeric and symbolic manipulation**: Because many programming solutions are based on the propper mixing of computations, the student needs basic skills on algebraic manipulations. A discussion on the importance of basic mathematic skills appears in [15, 30, 41, 101].
4. **Stepwise planning**. This category refers to student's ability to describe simple tasks in a step-by-step way, with thorough application of causal logic. The questioning can reveal levels of detail or abstraction, issues in thought expression, and misconceptions on algorithms. Some references to this ability appear in [128, 139, 174].
5. **Process analysis**: The student requires the ability to analyze and see the details behind an algorithmic process, in order to mechanically trace and debug programs. A discussion on the importance of interpreting algorithms and related issues appears in [13, 110, 127]. A discussion about hand-tracing appears in [8, 52, 80, 103, 110].

The cognitive origins of these capabilities have many different sources and develop through long periods. However, possessing this knowledge does not guarantee success in a particular course, but its absence seems to be usually a prelude for failing. It can be seen some relationship between deficiencies in the domains of the model and the problems listed in table 2.1.

1. **Reading comprehension**: Programmers need to read and understand many problem statements, requirements' specifications, and process descriptions. Related to this ability, is the one to write down information gotten from that context.

2. **Problem identification (abstraction)**: Programmers need to get essential information out of the problem context, and express it in the form of input, output, resources, and unknowns.

3. **Algebraic manipulation**: Programmers have to be skillful on basic algebraic manipulations because many programming solutions are based on the propper mixing of computations.

4. **Stepwise planning:** Regardless of the programming paradigm, the programmer requires to see the *big picture* of any problem represents, and methodologically decompose it into sub-problems; then figure out a *plan of attack* to solve each one.

5. **Process analysis**: The student requires the ability to analyze and see the details behind an algorithmic process, in order to mechanically trace and debug programs.

FIGURE 2.7. Model of Five Ability Domains (5-AD) for Algorithmic Problem Solving.

Chapter 3

Surveying Problem-Solving Ability

3.1 Introduction

As described in the previous chapters, fragile development of problem-solving skills is one of the main factors contributing to the struggle of programming students. Praxis has shown and the Lister-Fitzgerald team [110] has also recommended, that a problem-solving screening mechanism is needed to either perform specific studies on programming and problem solving, or to ponder its developmental level before they impact the teaching and learning process.

However, mechanisms adequate to the instructional needs of our field have apparently not yet been developed. As stated in the previous chapter, the instruments currently available do not address the areas or instances involved in computer programming, or do not provided the kind of information educators require to better understand and help their students.

This chapter discusses the design and implementation of an algorithmic problem-solving test that has mainly evolved from our work in the classroom.

3.2 Main Concerns on Assessing Problem Solving

The need for better assessment instruments and procedures in programming courses has motivated several studies [47, 48, 69, 117]. With respect to the problem-solving aspect of programming, the design of assessment instruments offers two major challenges.

First, assessing computing requires different types of questions to test basic information, analysis, and problem solving [180]. In the latter case, testing problem solving is difficult because of the skill subsets involved and their developmental nature, which tends to demand long questionnaires. This is particularly noticeable when multiple-choice questions are used because applicants are unable to show their thought processes behind particular answers [70, 85].[11]

Second, traditional testing models just report numerical estimates of applicant's level of dexterity on the knowledge or skill under testing [172], which does not reflect the intellectual capabilities or abilities involved. Information hidden in errors is disregarded, providing, therefore, very limited feedback about areas requiring improvement.

With exception of Fone's works [68, 69, 70], our literature review found no adequate model for qualitative assessment of problem solving. Fone has explored the possibility of developing automated assessment instruments that preserve advantages of traditional ones.

---

[11] According to Fone [70], factual knowledge (concepts) has static domains that can be acceptably represented when sampled by a small number of questions. Procedural knowledge (skills and application of concepts) has dynamic domains that are misrepresented if sampled with few questions.

3.3 The Study's First Phase

3.3.1 A Preliminary Survey

Aiming to survey problem-solving abilities of students starting introductory programming, Vasconcelos and Houlahan [168] developed a questionnaire with problems and questions drawn from the programming lifecycle and the problem-solving process (see table 3.1). The questionnaire items were either selective-response (multiple-choice), to quickly appreciate student's skills, or constructed-response (open answer), aiming to elucidate student's thought processes and concept understanding.

This paper-based questionnaire consisted of eleven problems, similar to common coursework exercises, and ten questions on programming and algorithm fundamentals; and it is described to detail in Appendix A.

Component	Question Types
1. Problem understanding	• Reading comprehension • Problem identification.
2. Problem analysis	• Summarizing information from statement. • Modeling situation and/or relations. • Functional decomposition of problems.
3. Solution design	• Perform structured planning. • Basic arithmetic. • Solve algebra word problems.
4. Solution review	• Interpretation of procedures. • Tracing and debugging pseudocode.

TABLE 3.1. Components of the Preliminary Survey.

3.3.2 Characteristics of the Pilot Application

- **Demographics**

The questionnaire was piloted by surveying 150 college students new to computer programming, 135 from "CS109: Introduction to programming" (September 2003) and 15 form "CS106: Algorithmic thinking" (January, 2004.) Because the diversity of the groups, attributes such as gender, ethnic or cultural backgrounds, or native language, were regarded as irrelevant.

- **Analysis of Results**

After the application of the instrument, a random sample of 20 questionnaires was analyzed, following a constructivist approach, i.e., trying to elucidate how the answer was constructed. Thus, besides knowledge or dexterity, we looked for error patterns and potential relations between different answers that could indicate skill fragility.

- **Feedback and Observation**

In addition to the survey, data was obtained from class observation and interviews. Such information served two purposes, first, get a deeper understanding of the information gathered with the survey, and second, obtain concrete opinions to improve the questionnaire.

3.4 Guidelines for Surveying Problem Solving

Besides the information originally expected, each answer in the survey provided important elements for future screening of problem-solving ability, and led to revisit assumptions behind the original survey. This exercise also served to pinpoint specific abilities requiring further attention and set the foundation for the second phase of our study.

3.4.1 Inventory of Common Pitfalls

The analysis of our first questionnaire revealed several problem-solving issues that can easily contribute to making programming instruction difficult. Examples include omitting or switching instructions, committing trivial arithmetic errors, neglecting quantifiers, failing to use algebraic relationships, or relying on unfounded assumptions. Such pitfalls were organized according to their occurrence within the 5-Ability Domain model and collected in an inventory (Table 3.2.)

It is worthy of mention that most items in our pitfall inventory correspond to problem-solving issues reported by several scholars (as discussed in Chapter 2, subsection 2.7.1), and related to several programming difficulties reported by Lahtien [105].

3.4.2 Categories of Questions

Our experience has confirmed that future surveys should be organized after the five-ability domain model involved in algorithmic problem-solving: *(i)* reading comprehension, *(ii)* problem abstraction, *(iii)* algebraic manipulation, *(iv)* stepwise planning, and *(v)* process analysis: tracing and debugging.

Ability Domain	Most Common Issues Detected
I **Reading comprehension**	• **Quantifiers:** quantifier in the answer does not correspond to the one used or implied in the text. • **Inference:** claim cannot be inferred from the text. • **External information**: answer includes information not provided by the text. • **Context**: meaning attributed to a given word does not correspond to text's context. • **Details**: omission of important details. • **Short rewritten**: Student write up does not summarize the text. • **Keywords**: Omission of most important words in the text.
II **Problem identification**	• **Missing information:** (a) the student did not detect that important information was not provided within the problem statement. (b) the student did not detect important information provided within the problem statement. • **Resources**: input data was not identified. • **Results**: output data was not identified. • **Input vs. output**: potential confusion between resources and outcomes. • **Generalizations**: failure to relate word problem, or concrete values, with generalized statement or algebraic relation.
III **Algebraic manipulation**	• **Wording**: problem wording confused student. • **Hierarchy**: algebraic hierarchy of operations was not applied correctly. • **Computation**: trivial arithmetic error. • **Steps out of order**: Steps were not followed correctly in a broken-down formula.

TABLE 3.2. Inventory of Common Pitfalls in Algorithmic Problem Solving.

Ability Domain	Most Common Issues Detected
IV Planning	• **Steps out of order**: Steps stated do not follow correct order. • **Redundancy:** several steps performing the same action. • **Oversimplification:** Problem not broken down into sub-problems. • **Logic:** (a) Causality not evident within steps provided, (b) failure to construct valid logic expressions. • **Abstractions:** Failure to handle abstractions properly. • **Loops**: (a) infinite loop, (b) incorrect number of iterations, (c) inability to express iterative procedures. • **Assumptions**: answer relays on invalid assumptions.
V Process analysis	• **Logic:** (a) Failure to observe causality between instructions, (b) failure to evaluate logic expressions. • **Failed Instruction:** instruction misinterpreted or omitted. • **Wording**: answer shows instruction wording confused student. • **Algorithm elements**: assignments, operations or initializations. • **States:** variable was not updated when instruction changed. • **Loops**: (a) infinite loop not detected, (b) incorrect number of iterations not detected, (c) confusion with the words *while* or *repeat*. • **Assumptions**: answer relays on invalid assumptions

TABLE 3.2. Inventory of Common Pitfalls in Algorithmic Problem Solving (cont.)

1. **Reading comprehension.** This category tests applicant's to objectively read texts, algebra word-problems, and instructions, by solving several questions without applying information not provided in the passage. Then, to verify if the reading was *understood,* the applicant's has to select a short passage, with different vocabulary, best describing the main one.
2. **Problem identification**. Questions within this category test applicant's ability to identify issues occurring within a given situation, abstract important data, organize information and summarize it within a problem statement. Also, it tests generalization of rules, usually expressed in the form of algebraic relations.
3. **Algebraic manipulation.** The questions within this category test the applicant's ability to perform simple arithmetic operations, either directly or in algebraic representation, the foreseen algorithm output, and to interpret word problems.
4. **Stepwise planning**. This category seeks to elucidate applicant's ability to describe simple tasks in algorithmic way —application of causal logic. The questioning can reveal levels of detail or abstraction, issues in thought expression, and misconceptions about algorithms.
5. **Process analysis**. This section tests the applicant's ability to work with sequences of simple instructions: reading algorithmic language, understanding the purpose of instructions, mechanical interpretation or hand-tracing, and proper utilization of algorithm properties. The questioning helps to elucidate consistency in application of logic, misconceptions in algorithms, and inadequate assumptions.

3.4.3 Interpretation of Answers

The first phase of our study showed the feasibility of analyzing questions with a constructivist approach, i.e., trying to elucidate how answers were constructed. Thus, rather than assessing knowledge or dexterity, we should look for clues that could indicate potential strengths or weaknesses to learn programming. The following sections shows and explains some examples of this approach.

- **Reverse Algebra Word Problem**

We call a "reverse algebra word problem" a word problem that moves from a given solution, or solution procedure, towards the problem that generated it. To solve this kind of problem, the student has to understand the sequence of operations stated and mentally map them onto one of the situations proposed.

Figure 3.1 uses a reverse algebra word problem to illustrate how even a small selective-response problem can provide a wealth of information. In this particular case, each answer is suitable, with just a subtle difference on its wording.

What would this sequence of instructions accomplish?

Step 1: Divide 100 by 24.

Step 2: Round that answer up to the next larger whole number.

a) Calculates how many gallons of gas are used to go 100 miles.

b) Calculates how many vehicles are needed to transport 100 people if every vehicle carries 24 people.

c) Calculates how many boxes will be completely filled with apples if 100 apples are to be put in 24 boxes.

d) All of the above.

FIGURE 3.1. A "Reverse Algebra Word Problem" with Partially Correct Options.

Let's review the problem carefully. If the instructions are followed correctly, the result is 5. At first glance, every option can seem valid, and thus, the best answer would be "d) all of the above." However, thorough reading reveals interesting details. For option (a), gas consumption is usually given without rounding —4.17 gallons were used in 100 miles. In the case of (b), five vehicles would be needed, four of them would be filled while the fifth one would carry just one student. In the case of the apples, option (c), the statement asks for the number of boxes with exactly 24 apples, which is four.

Thus, the only option strictly correct is (b), five buses. Nevertheless, any option provides information about applicants skills, from lucky guess to partial understanding, or deficient reading. The last option, "all of the above," might indicate either superficial reading or lack of understanding of the details involved in each option. The first option could be attributed to neglecting the rounding step, while the third one might indicate that attention was not paid to the key phrase "completely filled."

This "apparently obsessive" attention to details is a recurrent need within the programming activity: to ensure a running program, to detect minor errors in results, to troubleshot compiler errors, etc. [99].

- **Reading Comprehension**

Traditional reading comprehension tests can provide valuable insights on the reading ability required to create programs.

As discussed in Chapter 2, good programs result from fulfilling the specifications of requirements, usually a written document that has to be carefully

read and understood. The information appearing that document usually contains subtle details important to elaborate the product, as well as background data that could be not useful at all, and the programmer has to be careful to discern between both types. Figure 3.2 shows a reading that appears as project's prologue in a programming textbook,[12] and some of the options provided in the actual survey.

Instruction: Read this passage very carefully and mark as true (T) of false (F) the statements below. Select the answer that best matches the information given in the passage.

Prime numbers fascinate and frustrate everyone who studies them. Their definition is so simple and obvious; it is so easy to find a new one; multiplicative decomposition is such a natural operation. Why, then, do primes resist attempts to order and regulate them strongly? Do they have no order at all or are we too blind to see it? There is, of course, some order hidden in the primes. The Sieve of Eratosthenes shakes the primes out of the integers. First 2 is a prime. Now knock out every higher even integer (which must all be divisible by 2). The next higher surviving integer, 3, must also be prime. Knock out all its multiples, and 5 survives. Knock out the multiples of 5, and 7 remains. Keep on this way and each integer that falls through the sieve is a prime. This orderly if slow procedure will find every prime. Furthermore, as n goes to infinity, we know that the ratio of primes to non-primes among the first n integers approaches ($\log_e n$) */n. Unfortunately, the limit is only statistical and does not actually help in finding primes. [Wetherell's Etudes]*

a) The sieve takes advantage of Euclid's technique.
b) By using the sieve, all the resulting output are even numbers.
c) The passage describes the meaning of the prime numbers.
d) By using the sieve, the resulting output values are primes
e) The passage describes the technique to find all the prime numbers.
f) The passage describes one technique to find all the prime numbers.

FIGURE 3.2. Example of a "Reading Comprehension" Problem.

[12] Wetherell, C. *Etudes for Programmers*. Prentice Hall, 1978.

From the example, we can see that options (a) and (d) are checkpoints to observe whether the applicant has been attentive to both reading and answers. Option (b) serves to confirm if the distinction between odd and even numbers is known, and also if the text was interpreted correctly (i.e., the claim was stated in plural, but the technique only detects one even number, 2.) The meaning of prime numbers, option (c), cannot be inferred from the reading, and the difference between the last two statements is the quantifier (i.e., Eratosthenes' sieve is not the only way to find primes.)

Such level of detail while reading is very important throughout the whole programming lifecycle. Unfortunately, reading without sufficient attention is a recurrent issue pointed by specialists in problem-solving process, and always the first remark in problem-solving books and chapters.

- **Instruction Execution and Generalization**

The example shown in figure 3.3 states a sequence of arithmetic actions the applicant should perform in order to provide an answer (no options were provided).

What is the result of following these instructions?

Step 1: Think of a number, but keep it silently in your mind.	x
Step 2: Take your number and multiply it by 2.	$2x$
Step 3: Add 8 to the previous result.	$2x+8$
Step 4: Take the result in step 3 and subtract the number you started with.	$2x+8-x$
Step 5: What is the answer you got?	$x+8$

What is your answer? _____

FIGURE 3.3. A "Following Procedure" Problem with Symbolic Solution.

As it can be seen, the actions on each step are quite simple, they even might resemble some mathematical tricks that start with a number, request to perform some operations and lead to an specific value. However, in this exercise, the procedure does not ends with a concrete value. If the steps are followed correctly, in algebraic way, the answer should be a symbolic expresion, $x + 8$.

This kind of question can reveal if the student tends to think in concrete way, if the answer provided is a number, or in abstract way, if the answer provided is an algebraic expression. If no answer is provided, the applicant might have no idea of how to proceed, or be fearful of attemting any solution. In any case, this information can help the instructional process. For example, a student who provides an algebraic answer can be ready to deal with variables and generalized procedures, while a student who writes a number might require several concrete examples on variables before moving to generalized procedures.

In similar fashion, an answer omitted might indicate need for coaching: if the student runs into difficulties whenever making programs, special motivation could be required to prevent the learning process from stalling.

3.5 Automated Surveying

3.5.1 Paper-Based Lessons

The experience gained through our preliminary survey served to highlight that, even with selective-response questions, the students' answers could provide many insights on the problem-solving skills we were trying to screen. Furthermore, we could notice how some students' backgrounds or preconceptions are able to influence

(positively or negatively) the viability of some questions by triggering an attitude towards them, such as overconfidence, fear, or just the opposite. Some examples: a student already acquainted with Eratosthenes' sieve (fig. 3.2) can be less attentive to the reading, resulting in wrong answers; someone who had bad experiences with prime numbers might prefer to skip the question, and a good algebra problem-solver can get the correct answers easily without giving much thought to the problem-solving process.

To minimize the repercussion of such a predisposition, the questionnaire would require different questions for each category assessed, resulting in a longer questionnaire, or in different questionnaires according to each student needs. Also importantly, analyzing each questionnaire individually proved to be a very tedious and time-consuming process. If such instrument were intended for periodic application, results would hardly be on time to help students or preparing courses.

3.5.2 Key Features for a New Survey

This section outlines the main characteristics we considered in developing our new survey. They were intended to address the issues previously mentioned, while increasing the efficiency of the surveying task.

1. Use of automated-testing tools for increased efficiency of data gathering and returning.
2. Enable the survey to be online to allow easy, wide access, and reduce the need of specific software tools and platform dependency.
3. Develop a flexible architecture to allow easy upgrades and improvements of the survey.

4. Follow a progressive, experiential development (based on our class observations and peer advice) that will serve as foundation of next improvement iteration.

3.5.3 From Paper-Based to Web-Based

The first step in developing the online survey was to migrate the questionnaire from its paper-based format into a web-based equivalent. This was a two-folded task: (*i*) elaborate a front-end to display questions and get user's input, and (*ii*) implement a back-end monitor to verify answers and select new questions to be displayed.

The eleven original questions were transformed into HTML forms, one question per webpage, but taking care of preserving the respective answer style (selective or constructed response.) Additionally, a simple PHP test engine was implemented to verify correct selection of the question sequence and to check reliability of the data recorded (fig. 3.4).

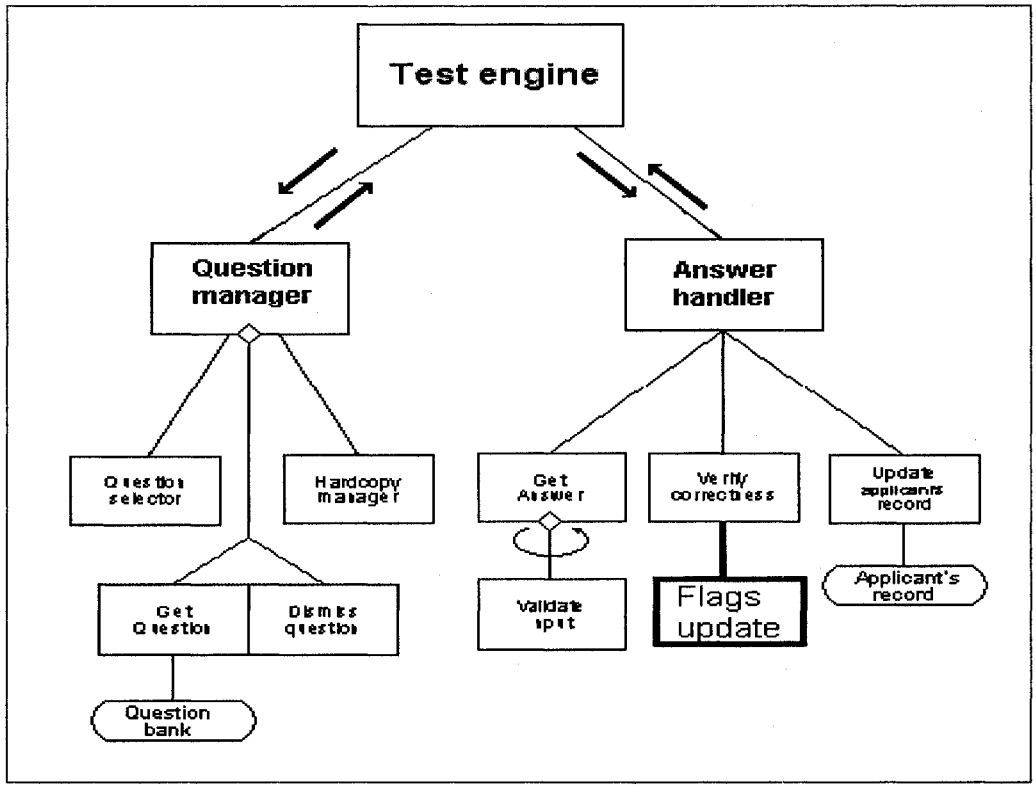

FIGURE 3.4 Main Functional Components of TAPSS 2.0 Testing Engine. The question manager, in charge of getting test items out of a question bank, works in conjunction with the answer handler, which gets user input for each question and scores it.

3.5.4 Adapting Question Types

The structure of the original constructed-response questions required modification to fit the constraints imposed by the multiple-choice format the online survey would present.

1. Questions originally stated as selective-response remained the same.
2. Questions originally stated as constructed-response were transformed into multiple-choice by using as options answers provided by students during the application of the original application.
3. Questions on process analysis and debugging (fig. 3.5a) have a somewhat

open nature: the issue has to be explained, although the error can be just marked down. Thus, these questions where split in two more: one asking to indicate what errors were found (fig. 3.5b), and another requesting to mark the faulty line, with possibility to add a brief explanation (fig. 3.5c.)

The following instructions were meant to display the decimal values of the sequence 1/1,1/2, 1/3, 1/4, 1/5, however it does not work correctly.

```
Set num to 0
While num < 5
        Compute dec = 1 / num
        Display dec
        Add 1 to num
End While
```

FIGURE 3.5a Example of a Debugging Problem.

What kind of problem(s) did you find?

a) Division by zero.
b) No result displayed.
c) No data input.
d) Never ending loop.
e) Never starting loop.
f) Incorrect number of iterations.
g) Wrong initialization.

FIGURE 3.5b Possible Bugs in the Algorithm of Figure 3.5a.

Mark and correct any faulty instruction

a) Set num to 0 _____
b) While num < 5 _____
c) Compute dec = 1 / num _____
d) Display dec _____
e) Add 1 to num _____
f) End While _____

FIGURE 3.5c Algorithm with Options to Mark Errors and Write Corrections.

4. Problems on planning are creative in nature and cannot be effectively tested with multiple-choice questions. Thus, we decided to present several stepwise answers to the problem (figure 3.6). The difference between options is interchanged steps or different solutions.

Make a stepwise procedure to compute the following sequence up to six terms: 1, 4, 27, 256, 3125, ?

Procedure 1

1. Let a be the first natural number
2. Multiply a, a times.
3. Let b be the result of the previous step.
4. Increase a to the next natural number
5. Repeat steps (2) to (4) five more times.

Procedure 2

1. Let a be the first natural number
2. Let b the result of multiplying a, a times.
3. Write down b
4. Increase a to the next natural number
5. Repeat steps (2) to (4) five more times.

Procedure 3

1. Let a be the first natural number
2. Let b the result of multiplying a, a times.
3. Write down b
4. Increase a to the next natural number
5. Repeat steps (2) to (4) four more times.

FIGURE 3.6 Example of a Multiple-Choice Question on Planning.

3.5.5 Skill Tracking Variables

The test engine (fig. 3.4) was designed to monitor several values: answer correctness, pitfall occurrence, and skill performance. Tracking variables were drawn from two main sources: the pitfall inventory (table 3.1) and an expanded version of our ability domain model (table 3.3.)

Each pitfall and sub-domain corresponds to specific variables (counters) that were weighted after user answers. Sub-domain counters are updated according directly to answer's correctness while pitfall variables are updated only with incorrect answers (otherwise the pitfall did not occur).

For example, a correct response on the question of figure 3.3's would indicate abilities to (*i*) read and follow instructions, (*ii*) identify a problem out of a sequence of instructions, (*iii*) correct use of variables and algebraic expressions, and (*iv*) no pitfall committed. On the contrary, a mistake answering figure 3.7's question would automatically update the computation pitfall. However, because lack of attention might not be the only cause of the error, so, sub-domain variables on reading and algebra word problems are also updated negatively.

Ability domain	Code	Sub-domain
I Reading Comprehension	[11]	Problem Statement.
	[12]	Algebra word problems.
	[13]	Procedures in narrative style.
	[14]	Procedures in stepwise form.
	[15]	Pseudocode.
II Problem Identification	[21]	Problem statement.
	[22]	Problem context.
	[23]	Algebra word problem.
	[25]	Instruction sequence.
	[27]	Problem type.
III Arithmetic & Logic	[31]	Algebra word problem.
	[32]	Evaluation of arithmetic expression.
	[33]	Operators hierarchy and laws.
	[36]	Iterative operations.
	[37]	Variables as abstraction.
	[38]	Boolean expressions.
	[40]	Conditions.
IV Planning	[43]	State the sequence of a set of instructions.
	[44]	Elaborate a solution and compare with others.
	[45]	Identify problem type.
	[47]	Assignment and expression evaluation.
V Process Analysis	[52]	Following instructions.
	[53]	Hand-tracing.
	[54]	Prediction of results.
	[55]	Debugging.
	[57]	Interpreting instructions in different formats.
	[59]	Variable as memory cell.
	[60]	Array as entity.

TABLE 3.3. **Inventory of Sub-Domains of Algorithmic Problem-Solving Ability and Skill Tracking Codes.** These sub-domains where obtained by observing how each ability domain is involved in different aspects of the problem solving process.

Your younger brother is planning a sleepover with 4 friends. Your mother told him to buy 2 hot dogs, 3 candy bars and something to read, for himself and each guest. He also needs some soda, and knows that 1 liter of soda is enough for 3 kids. How much food will he buy at the store?

a) 2 Hotdogs, 3 candies, 1 soda, 5 Comics.
b) 10 Hotdogs, 15 candies, 2 sodas, 5 Comics.
c) 10 Hotdogs, 15 candies, 1 soda, 6 Comics.
d) 12 Hotdogs, 18 candies, 2 sodas, 5 Comics.

FIGURE 3.7. Example of an Algebra Word Problem.

3.5.6 Question Calibration

All the items in the questionnaire were calibrated (reviewed and adjusted) according to these three factors: the 5-ability domain model and its sub-domains, multiple valid answers, and oversampling.

First, we ensured that each question belonged to a specific category within our model, and that all the options were properly aligned to pitfalls and sub-domains. (See example on subsection 3.5.5.)[13]

Second, despite traditional guidelines for multiple-choice question design – only one correct option in the answer set [35, 84], we implemented questions with multiple valid answers (see figure 3.1) aiming to recognize partial understanding and developmental levels of ability. (According to their webpage, the profiling company Skillprofiller successfully uses this technique [148].)

Third, oversampling, a technique also used by Fone [70] to improve the design of multiple choice question tests. According to Fone's semiotic point of view, each question is like an instrument that gets some information (a sample) of

[13] Concepts and experimental questions did not appear in the final survey.

student's knowledge on a specific topic/objective. However, if such instrument fails to match topic or student's way to express it, the test fails to verify what he/she really knows. But, if the instrument is prepared to be redundant (via oversampling), the likelihood of missing information because one question (in many) failed as instrument, is reduced (fig. 3.8).

In an analogous way, each question in our online survey was calibrated to regard the different ability sub-domains that it could measure. For example, the question in figure 3.3 spans: reading comprehension of a problem statement and a procedure in stepwise form, identify an algebra word problem and the mechanics of an instruction sequence, correct use of variables and evaluation of algebraic expressions, following instructions and hand-tracing.

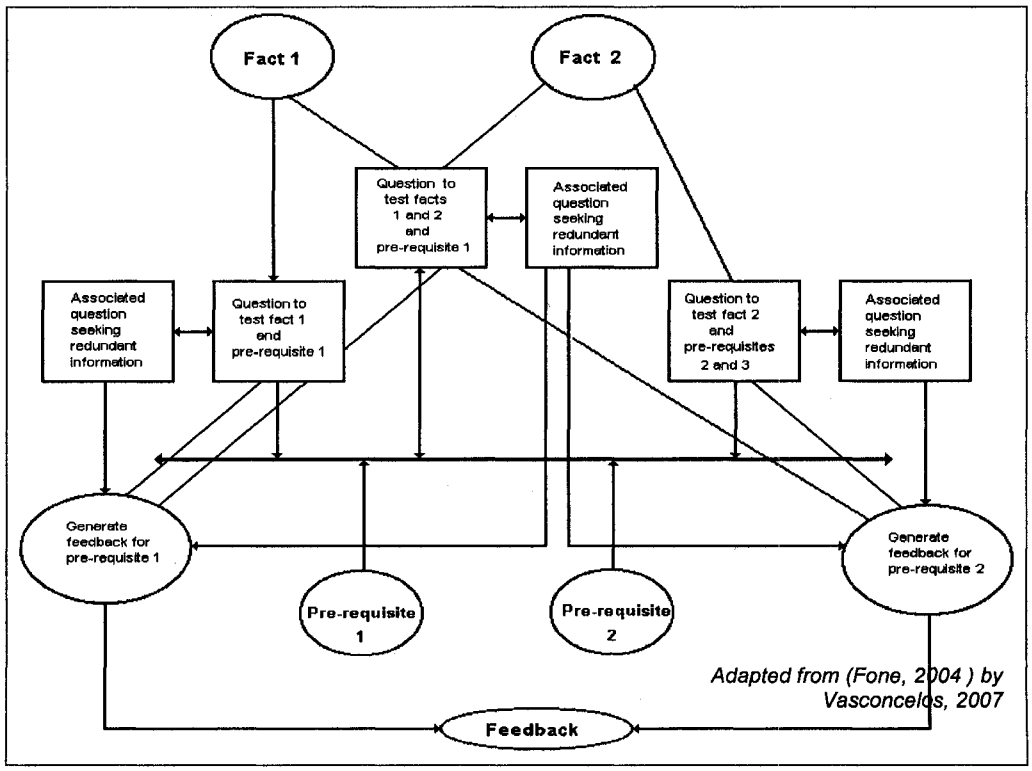

FIGURE 3.8. Model of Linked Questions to Oversample Skill Domains (Adapted from [70].)

Because all the questions are structured this way, a negligible mistake in one particular question would be compensated by subsequent questions.

3.5.7 Adapting Test Engine to Question Type

Subsection 3.5.4 discussed the need to modify the open-ended nature of some questions to simplify the implementation of a multiple-choice questionnaire. Although changes were easy to carry out in the front-end (the HTML provides rich set of elements to create a variety of forms), they require creative work to transform the test engine in a flexible back end.

Adapting the test engine to different question styles required us to standardize types and inventories, dissociate answer information from question file or test engine, and set a reliable structure for each one.

1. To handle the inventories, a numeric code was associated to the elements of tables 3.1 and 3.2.
2. For each question two files were specified, one for the HTML form, and one for the descriptor containing solution, question type, and associated data.
3. Each question solution file was given a structure according to its type, but easy to decode by the test engine.

```
(a) #Descriptor file for question: q1003
(b) QType=1&Nopts=4&Right=3&Incorrect=2
(c) f1=303&f2=000&f3=000&f4=301
(d) r1=33&r2=25&r3=14&r4=52
(f) w1=25&w2=52
```

FIGURE 3.9. Descriptor of a Simple MCQ Problem.

For example, for a simple multiple-choice question (fig. 3.9), the descriptor file consists of five vectors:

a) Identification line (1 element): Question identification.

b) Answer information (4 elements): Question type, number of options, number of the correct option, and number of the incorrect option.[14]

c) Pitfall vector (same elements as number of options): If the question is not answered correctly, the pitfall code corresponding to the option selected serves to update the corresponding variable.

d) Ability sub-domains I (variable length): If the question is answered correctly, all the sub-domains which code is listed are updated positively.

e) Ability sub-domains II (variable length): If the question is answered incorrectly, all the sub-domains which code is listed are updated negatively.

Table 3.4 summarizes the question types and their respective file structure.

Question Type	Descriptor Structure
1 **MCQ**	• 1 correct answer. • 1 wrong answer. • *n* viable answers, each associated with a specific pitfall flag. • 1 set of ADs to be updated if correct answer is selected. • 1 set of ADs to be updated if wrong answer is selected.
2 **Checklist**	• *n* correct answers. • *m* incorrect answers. • *l* viable answers, each associated with a pitfall code.

TABLE 3.4. Descriptors According to Question Type.

[14] The "incorrect option number" is a legacy feature. It was kept for compatibility between the test engine and old questions and techniques.

Question Type	Descriptor Structure
3 **YNQ**	• n Y/N questions, each associated with pitfall code. • 1 set of n m-tuples: o correct answer (true or false). o 1 set of ADs to be updated if correct. o 1 set of ADs to be updated if incorrect. o 1 pitfall code if incorrect.
4 **Checklist II** (With open explanation)	• Possibilities when debugging a program: o No error, no detection, correction not attempted → OK. o Error, no detection, correction not attempted → Possible distraction. o No error, no detection, correction attempted → Debugging problem. o Error, no detection, correction attempted → Possible distraction. o No error, detection, correction not attempted → Unlucky guess. o Error, detection, correction not attempted → Lucky guess or unable to correct. o No error detection, correction attempted → Debugging problem. o Error, detection, correction attempted → OK.
5 **Semi-open questions**	• Answers easy to be parsed: numerical results, one letter/word input, etc.
6 **Open questions**	• Questions to be read manually: o For feedback o To test new questions

TABLE 3.4. Descriptors According to Question Type (cont.)

3.6 Enhanced Multiple-Choice Questions

3.6.1 Rating Confidence on Answer

To get more insights regarding student's abilities, the survey requires each answer to be rated according with the confidence the student has on in it (fig. 3.10).

This technique, suggested by Blunck (2007) and described by Fone in [68], helps the surveying process to easily appreciate how suitable are student's mental models with respect to different problem-solving aspects.

What results of following these instructions?

Step 1: Think of a number, but keep it silently in your mind.
Step 2: Take your number and multiply it by 2.
Step 3: Add 8 to the previous result.
Step 4: Take the result in step 3 and subtract the number you started with.
Step 5: Write down your answer.

Select one option: (a) 13 (b) x (c) 8 (d) ß+8 (e) $2x$+8

How confident do you feel with your answer?

(a) I just guessed (b) I had some idea (c) I knew it

FIGURE 3.10. An MCQ Problem Enhanced with Confidence Level.[15]

[15] The original question had a constructed-response structure (fig. 3.3). It was transformed into selective-response question by providing some options: two concrete values and three algebraic expressions. To prevent easy recognition of the correct answer (d), the traditional algebraic letter x was changed by the Greek symbol ß.

For example, if the student claims to know the answer, and it was correct, then, the testing process can be regarded without problem. A similar case occur if the student claims to have guessed, and it was incorrect.

However, claiming that the answer was known while it was incorrect, then the testing process has detected a problem, from a misconception to overconfidence. In the opposite side, assuring that a correct answer was just guessed points to a need of providing reassuring feedback to stimulate and raise confidence.

Furthermore, as discussed in Chapter 2, being aware of the confidence behind every action can be very helpful while writing programs [74]. For example, a common situation is proposing a procedure that solves the situation stated but that seems unsuitable in other cases. Many students tend to be deterred from solving problems and making programs while facing with such uncertainty. Nevertheless, if taught how to work in spite of it, and how to document it, could have positive consequences and help the students during the review stages of the program lifecycle.

3.6.2 Option to Skip Question

Traditionally, skipping multiple choice questions is discouraged, regarded as an erroneous answer, or completely prevented. However, forcing the student to select an answer in spite of uncertainty belittles the purpose of the assessment. The opposite case is similar; a question left unanswered provides no information at all about the reasons behind such decision like lack of time, fear of error, misunderstanding, uncertainty or fragile knowledge.

What results of following these instructions?

Step 1: Think of a number, but keep it silently in your mind.

Step 2: Take your number and multiply it by 2.

Step 3: Add 8 to the previous result.

Step 4: Take the result in step 3 and subtract the number you started with.

Step 5: Write down your answer.

Select one option: (a) 13 (b) x (c) 8 (d) ß+8 (e) $2x+8$

Why are you skipping this question?

(a) The answer does not appear among the options.

(b) I don't know the answer.

(c) I don't understand the question at all.

(d) Other reason: _____

FIGURE 3.11. An MCQ Problem with Skipping Option.

There are several aspects to consider the possibility of skipping questions.

1. The questionnaire might be imperfect; questions or options might be unclear or plain wrong. Skipping such questions, providing proper explanation, can be very helpful to the test designer.
2. A questionnaire can be completely correct but a student might indicate the opposite; such situation might be an indicator of reading problems or even lack of interest.
3. Skipping a question can be indicator of critical thinking. A student that knows the correct answer and skips it might have better developed decision-making skills than someone that only selects the answers provided.

In similar fashion to the discussion in the previous section, being aware of the reasons behind every action, or inaction, is very important while making

programs. If a part of the solution process is unknown, it can be better to document the issue and skip it, than become stuck on it. In a different case, not taking the options at face value can be helpful during the initial tests of the program. (Some students tend to accept any result from an incorrect program just because the program appeared in a book, was given by the instructor or was produced by computer.)

3.7 Web-Based Survey Prototype

A web-based surveying prototype, including all features detailed on sections 3.5 and 3.6, was released for testing on October 25, 2007, and it is located at http://www.cs.jhu.edu/~jorgev/survey.

3.7.1 Questionnaire

To simplify the implementation, the current format is of fixed-item length but, in most cases, each item is selected from a small question bank. The main purpose each item in the questionnaire is described in table 3.5.

Ability Domain	No.	Question type	Description
1 Reading Comprehension	1	**Passage level**	Testing reading comprehension on a generic content passage. (Main characteristics are described in a GRE preparation guide cited in [103].)
	2	**Word level**	Testing reading comprehension on a specific content passage. (Described in [103] and adapted to common issues with matrices.)
2 Problem Identification	3	**Summarizing**	Identify best sentence summarizing a reading. (Question aimed to test ability to abstract a problem out of its context. Designed after McCracken's observations [117].)
	4	**Reverse word problem translation**	Identify the purpose of a sequence of arithmetic operations. (Question aimed to test ability to match a general solution with a concrete problem.)
3 Algebra & Logic	5	**Word problem translation**	State a formula, or a short sequence of arithmetic operations, to solve an algebra word problem. (Question designed by Mayer [114] to test problem translation skill.)
	6	**Word problem solution (L)**	Typical algebra word problem; solution can be deduced logically.
	7	**Word problem solution (A)**	Typical algebra word problem; solution requires arithmetic operations. (Question designed by Mayer [114] to test problem translation skill.)
	8	**Following procedure (A)**	Trace a sequence of algebraic operations that leads to an algebraic answer. (Question aimed to test ability to follow a procedure in abstract way.)
	9	**Boolean expressions I**	Simple question on inequalities and conditions.
	10	**Boolean expressions II**	Simple question on logic operators and truth tables.

TABLE 3.5. Inventory of Skills Tested with TAPSS 2.0. Some aspects of the questionnaire items were adapted from [103, 114].

Ability Domain	No.	Question type	Description
4 Stepwise planning	11	**Symbolic sequence**	State steps to solve an arithmetic problem. Different sequences are provided, and the student has to select one that solves the problem. (Selective-response question aimed to test ability for symbolic stepwise planning.)
	12	**Statement sequence**	State in English (pseudocode) the steps to perform a specific task, for example, produce a numerical sequence. (Selective-response question aimed to test ability for descriptive stepwise planning.)
5 Procedure comprehension	13	**Debugging pseudocode I**	Finding errors. (Question aimed to test ability for debugging a solution. Designed after common program debugging questions.)
	14	**Debugging pseudocode II**	Correcting errors. (Question aimed to test ability for debugging a solution. Designed after common program debugging questions.)
	15	**Following instructions (S)**	Indicate the result obtained after following instructions in narrative way (symbolic). (Question designed by Mayer [114] to test procedure comprehension skill.)
	16	**Following procedure (N)**	Indicate the result obtained after following instructions in narrative way (numerical). (Question designed by Mayer, 1986, to test procedure comprehension skill.)
6 Thinking skills	17	**Critical thinking & details**	Multiple-choice question with no correct answer provided. (Question aimed to test attention to details —lack of correct answer— and critical thinking skills —how to proceed.)

TABLE 3.5. Inventory of Skills Tested with TAPSS 2.0 (cont.)

3.7.2 Front-End

The survey's front-end relies on a web browser's capabilities to display each question and related options as webpages. User's input is handled through the mechanisms of HTML forms, while question events (numbering, confidence request, skipping confirmation, etc.) are supported by Javascript snippets embedded in each webpage. There is a minimal tracking sub-system that relies on *cookies* and holding user id, question number, and survey status. Figure 3.12 depicts how the webpage changes state according to the interaction with the user. For example, every time the test engine issues a question, it starts at state Q_0. Then the user is given two options, either select an answer or skip the question. In the first case, the webpage changes state to Q_1 (question answered) and, automatically, goes to state F_1 (confidence request); in the second case, the page moves to state F_2, in which the user either confirms the action or cancels the skip.

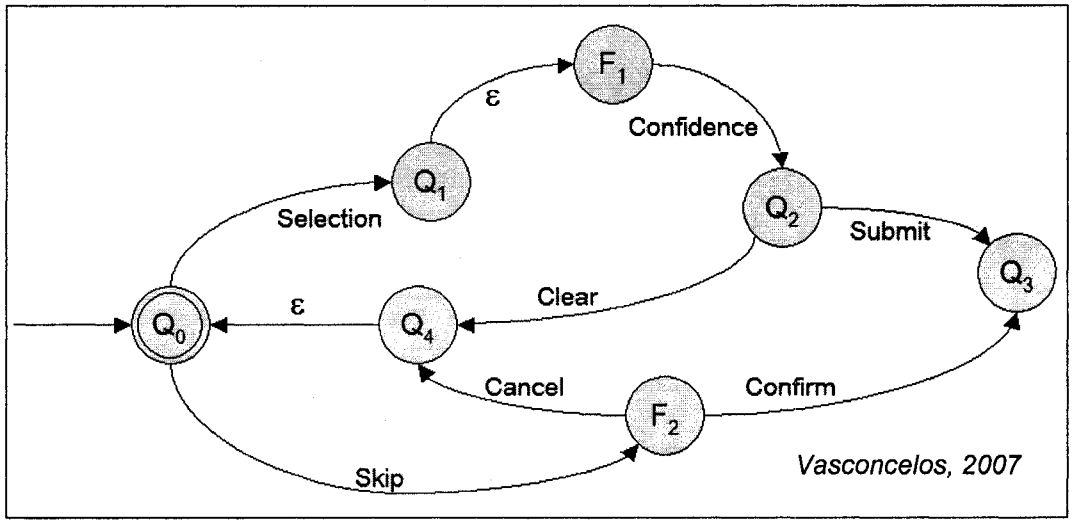

FIGURE 3.12. State-Diagram Model of TAPSS 2.0 Front-End. Q_0 = New question, Q_1 = Answered question, Q_2 = Answered question with confidence level, Q_3 = Question ready for evaluation, Q_4 = Options reset. F_1 = Input request for confidence level, F_2 = Input request for skipping explanation.

3.8 The Study's Second Phase

This section discusses the pilot application of the online Test of Algorithmic-Problem Solving Skills, TAPSS 2.0, administered to a small group during the period October 25 to November 19, 2007. The information collected served to verify data integrity, software reliability and flexibility, and to validate interpretation.

3.8.1 Demographics

The control group consisted of fifteen applicants[16] with college-level studies and programming experience known prior to the application of the test. Attributes such as gender, ethnic or cultural backgrounds, and native language, were regarded as irrelevant for this part of the study. Applicants were classified within three subgroups according to their level of expertise in programming:

- **Non-experienced**: This subgroup consisted of four people with virtually no knowledge of programming[17] and whose interaction with computers was limited to mainstream applications. Members of this subgroup had no plans of coursing programming in the near future.
- **Literacy**: This subgroup consisted of six persons who were well versed with information technologies and somewhat acquainted with programming. At the time, people in this subgroup were involved in the field of information technology or had a formal programming course several years ago.

---

[16] Group size was deliberately kept small to allow manual review of each questionnaire submitted, as well as every stage within the surveying process.

[17] Some people in this subgroup might have attended a training course in programming before 1997. However there was minimum learning and the skill was never practiced.

- **Experienced**: This subgroup consisted of five persons that were already developing software or information systems, and had several years of experience.

3.8.2 Scoring

- **Questionnaire**

Each question described in table 3.4 was assigned one point: +1 if answered correctly and –1 otherwise. Skipped questions did not receive any points. This raw data was filed and used to compute all other results described in Chapter 4.

- **Confidence Factor**

The confidence rate asked the applicant after answering each question was used to weight raw scores (table 3.6.) For example, if the applicant was guessing, the answer was disregarded because it does not actually reflect strength or weakness. On the contrary, if the applicant was sure, the value of answer was magnified to reflect either a strong skill or a potentially serious misconception.

Answer		Confidence		Adjustment	
Status	Raw score	Level	Multiplying factor	Interpretation	Score
Correct	1	Sure	2	Very good	2
Correct	1	Unsure	1	Good	1
Correct	1	Guess	0	Disregard	0
Incorrect	-1	Guess	0	Disregard	0
Incorrect	-1	Unsure	1	Attention	-1
Incorrect	-1	Sure	2	Misconception	-2

TABLE 3.6. Score Adjustment According to Confidence Level.

- **Cross-Domain Skill Tracking**

In addition, associated sub-domains (see subsection 3.5.5) were updated accordingly to answer correctness. Main sub-domains related to each question are shown in table 3.7. (Skill tracking codes are listed in table 3.2.)

No.	Question type	Skill Tracking Codes
1	Passage level	11, 12, 13, 21, 22, 23, 57, 61
2	Word level	11, 21, 22, 60
3	Summarizing	11, 21, 22
4	Reverse word problem translation	14, 27, 25, 32, 33, 51, 61
5	Word problem translation	12, 31, 33, 37, 47
6	Word problem solution (L)	12, 31, 54
7	Word problem solution (A)	12, 31, 32, 54
8	Following procedure (A)	14, 25, 37, 47, 53, 57
9	Boolean expressions I	12, 23, 37, 40
10	Boolean expressions II	12, 23, 38
11	Symbolic sequence	11, 14, 22, 27, 32, 33, 43, 44, 47
12	Statement sequence	15, 27, 32, 36, 37, 43, 44, 45
13	Debugging pseudocode I	11, 15, 21, 32, 37, 40, 47, 54, 55, 57 59
14	Debugging pseudocode II	23, 36, 37, 40, 47, 53, 55, 57, 59
15	Following instructions (S)	13, 21, 52, 53, 60
16	Following procedure (N)	14, 32, 37, 52, 53, 59
17	Critical thinking & details	61

TABLE 3.7. Main Sub-Domains Associated to Questions in TAPSS 2.0.

- **Cross-Question Skill Tracking**

Table 3.8 shows the questions that were regarded to integrate skill subsets. Those questions were averaged to compute a general score for each skill subset involved in the survey.

AD	Skills	Linked Questions
1	General Reading	1, 3
2	Problem identification	3, 4
3	Word problem translation	5, 6, 7
	Arithmetic operations	4, 6, 7
	Boolean logic	9, 10
	Handling Variables	5, 16
4	Planning	11, 12
5	Reading pseudocode	13, 14
	Following instructions	8, 15, 16
	Handling arrays	2, 15
6	Dealing with abstraction	3, 8
	Details & Critical Thinking	2, 8, 17

TABLE 3.8. Linked Questions to Oversample Skill Domains.

3.8.2 Data Verification

To verify data recording, question scoring, as well as integrity of files in use, each answer submitted to the system was hand reviewed and analyzed following the scoring process previously described and the model implemented in TAPSS 2.0.

3.8.3 Question Validation

To appreciate how well the questions matched the intended skill (see table 3.4), each answer was related with the corresponding confidence level and the programming experience of the test-taker. In addition, the construct behind most questions was also reviewed through available, particularly reading comprehension [103], problem translation and procedure comprehension skills [114].

3.8.4 Data Interpretation

Answer correctness was used as the primary confirmation of presence of the skill for which it was intended. Pitfall and sub-domain tracking helped for a more detailed review, which can point for recalibration needs.

3.8.5 Application Feedback

Some applicants were interviewed about their experience with the questionnaire, the surveying system, and also on their opinion about the results. Most meaningful comments appear in the next chapter.

Chapter 4

Results and Discussion

4.1 Summary of Results

Table 4.1 shows a comparative of skill strength for the subgroups in the study.

AD	Skills	No experience ($N=4$)		Literacy ($N=6$)		Experienced ($N=5$)	
		Score/Strength		Score/Strength		Score/Strength	
1	General Reading	-0.2	W	0.8	S	0.8	S
2	Problem identification	0.2	B	0.1	B	0.1	B
3	Word problem translation	0.1	B	0.5	A	0.5	A
	Arithmetic operations	0.2	B	0.4	A	0.4	A
	Boolean logic	0.1	B	0.9	S	0.9	S
	Handling Variables	0.0	B	0.2	B	0.2	B
4	Planning	0.0	B	0.6	A	0.6	A
5	Reading pseudocode	0.0	B	0.5	A	0.5	A
	Following instructions	-0.2	W	0.1	B	0.1	B
	Handling arrays	-0.1	B	0.6	A	0.6	A
	Dealing with abstraction	-0.4	W	-0.1	B	-0.1	B
	Details & Critical Thinking	-0.2	W	0.	A	0.3	A

TABLE 4.1. Summary of Skill Strength According to Programming Expertise.

Codes: S-strong, A-Acceptable, B-baseline, W-Weak.

The information on table 4.1 appears graphically in figure 4.1.

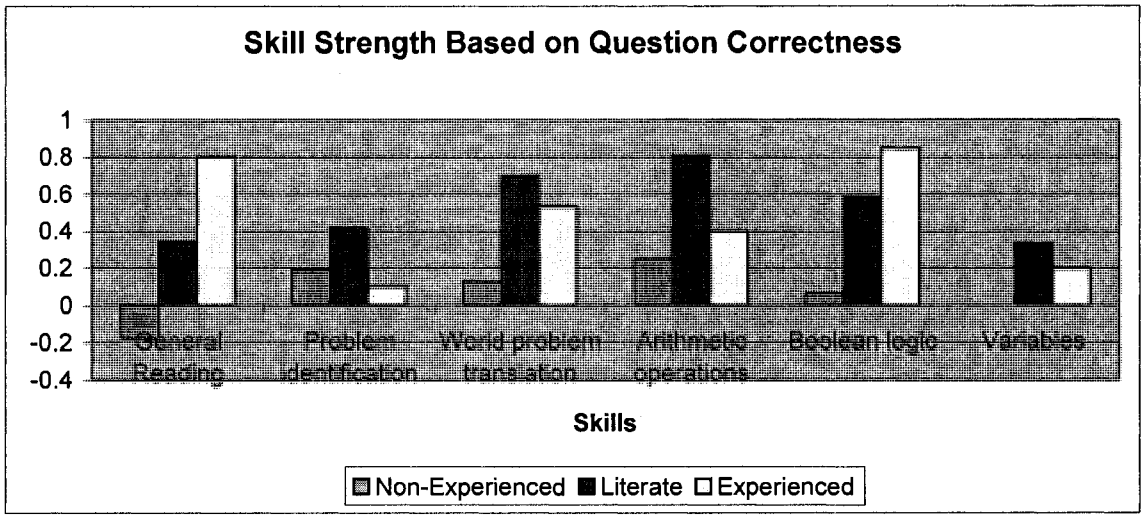

FIGURE 4.1. Skill Strength According to Level of Programming Expertise.

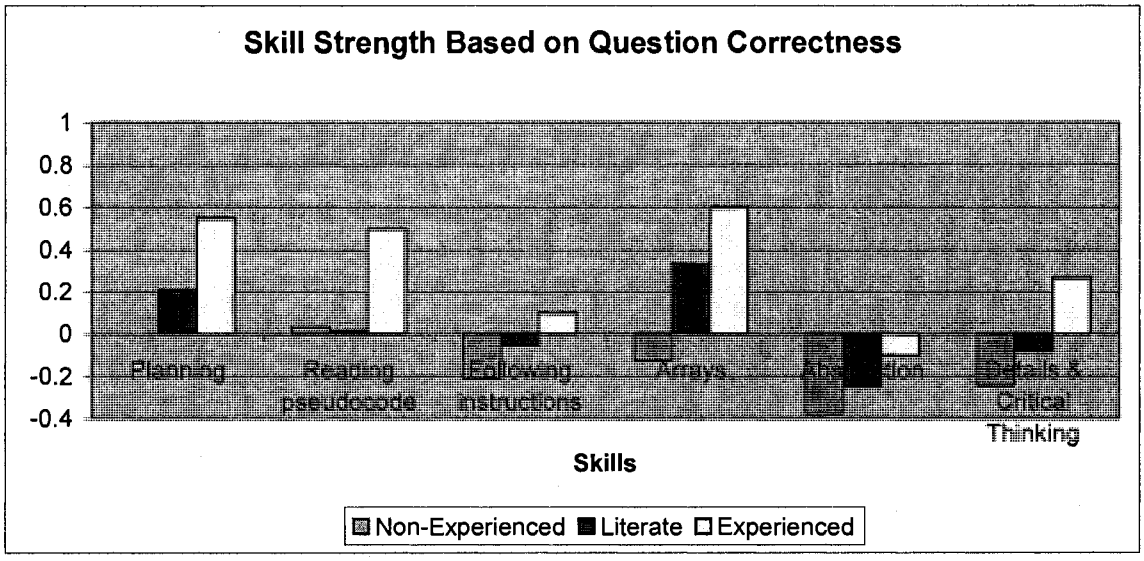

FIGURE 4.1. Skill Strength... (cont.)

4.2 Analysis of Results

Charts in figures 4.3 through 4.6 show results from the software beta-testers, who used the TAPSS 2.0 prototype during its trial period (10/25 to 11/19). Data appears organized according to levels of programming expertise: non-experienced, literacy,

and experienced. Questionnaire items (table 3.4) are displayed in the horizontal axis while answer correctness appears in the vertical one.

Correctness is represented with a value between 0 and 1.0 —from absolute mistake to perfect answer. Skipped questions are represented with negative units. The scores from problems with sub-questions, items #1, #13 and #14, were computed by dividing the number of correct answers by the total number of sub-questions. Thus, a score of 1.0 would be obtained if all sub-questions were answered correctly.

4.2.1 Survey Overview

Figure 4.2 presents a general view of participants' results, organized according to programming experience. In this chart, answer correctness is presented as average of all the correct scores on the corresponding question. Incorrect questions were not regarded, and skipped ones were assigned a negative value to better appreciate them.

Simple inspection shows that most skipped questions occurred within the non-experienced subgroup. The literate subgroup tended to skip problems on planning (questions #11 and #12) and procedure debugging (question #13).

The performance of the experienced subgroup was consistently strong throughout the test. In contrast, the performance of the literate subgroup faded noticeable on the domains of planning (questions #11 and #12) and process analysis (questions #13 through #16).

Answers with low level of correctness in both, experienced and literacy subgroups, might indicate either an unsuitable question or a very skill-specific one.

Such seems to be the case of question #3, a problem on information abstraction, and question #8, a following-procedure problem with symbolic answer.

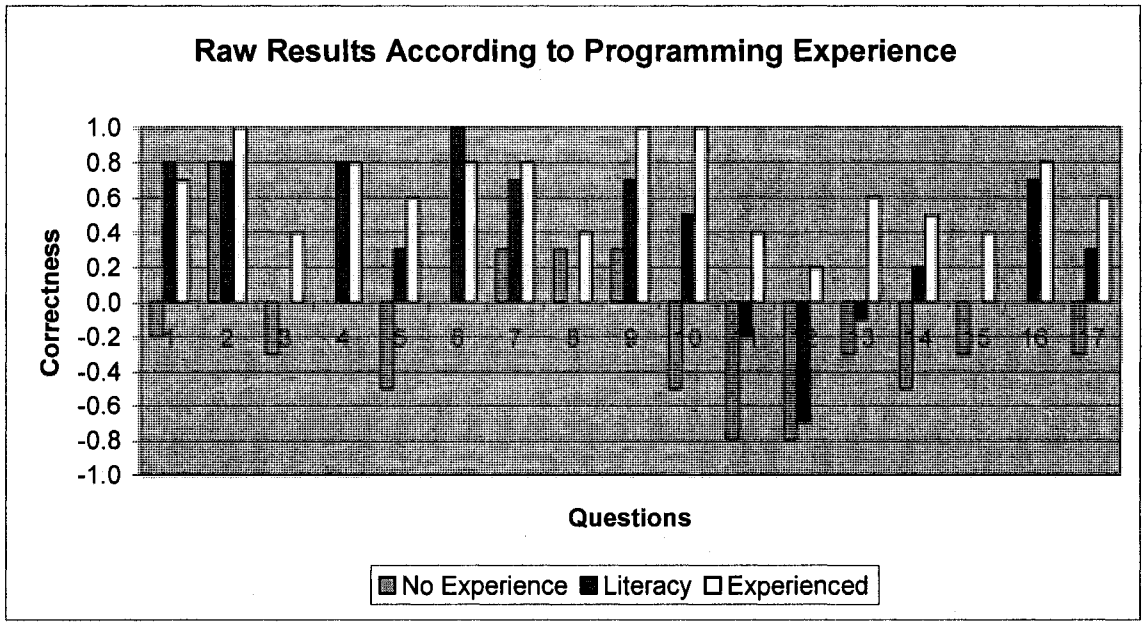

FIGURE 4.2. Average Results from TAPSS 2.0 Pilot Administration. On average, experienced participants were strong and did not skip questions. Literate participants were almost as strong as experienced ones, except in planning and process analysis (questions 11 to 16) where several questions were skipped. The non-experienced subgroup is in clear disadvantage.

Results on question #3 can be interpreted as confirmation of the McCracken's group observation "the most difficult part for students seemed to be abstracting the problem to be solved from the exercise description." [117, p.133] With respect to question #8 (fig. 3.10), a problem the author has repeatedly used when teaching introductory programming[18] as item for pre-testing and discussion. In our experience, most students give a numeric (concrete) answer, and only very few

[18] The actual version the author has used in programming courses is shown in figure 3.3.

talented students —not necessarily experienced in programming— provide an algebraic (abstract) one.[19]

Because both questions deal with abstraction, in its two main meanings — detail removal and non-concrete thinking, the results can be aligned with Kramer's observations on the issues of abstraction in computer education [100, 2007].

With respect to the last question (#17), testing attention to details and critical thinking, it is clear that the experienced subgroup had advantage over the other two.

4.2.2 Results from the Non-Experienced Subgroup

Figure 4.3 presents results from four participants with no programming experience. This subgroup had six members initially, but two people decided two quit the survey after reading the first question. The brief comments they provided pointed to test/topic anxiety.

In general, participants either skipped questions or guessed answers (see fig. 4.6), which does not provide enough data that could lead to infer applicants' strengths or potential learning hazards. In this sense, quitting is a clearer indication of a major learning hazard: programming demands tenacity and ability to persevere, attributes also expressed by students in ChMura's study [37, p.56a].

[19] The author is somewhat surprised by the results of question #8, particularly in the case of experienced programmers. When the original question was transformed into a MCQ, the author was unwillingly expecting that the options would work as hints for the applicants.

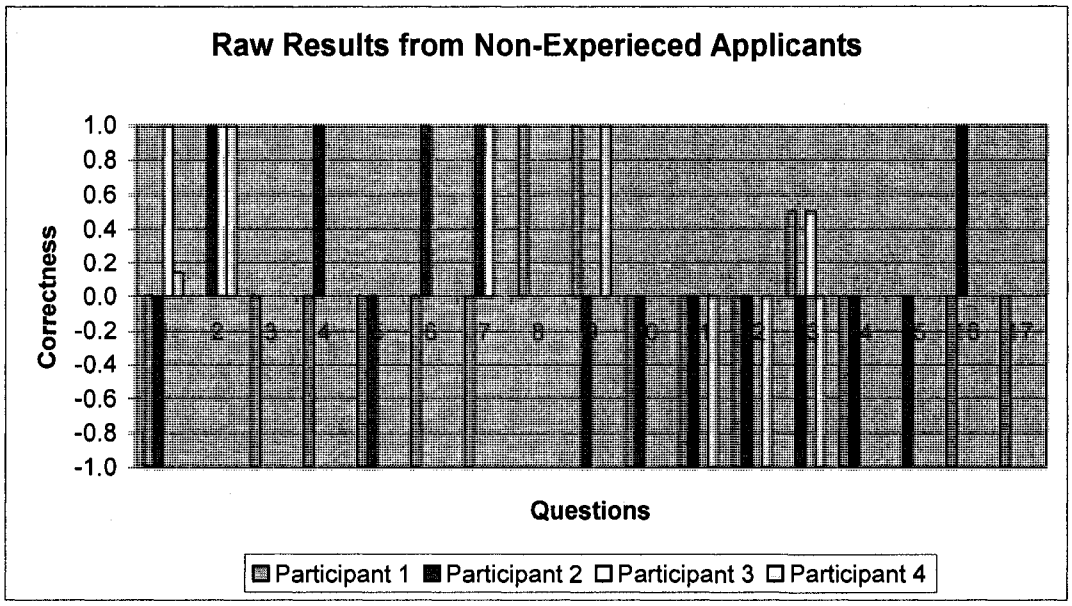

FIGURE 4.3. Results from Applicants with no Programming Experience. A missing bar indicates a wrong answer in the specific question where it should appear. Bars with negative value represent skipped questions.

4.2.3 Results from the Literate Subgroup

Figure 4.4 presents the results from six participants with programming experience at literacy level. Simple inspection shows that most skipped questions came from the same applicant. Also, it can be seen that participants faced most difficulties on planning (questions #11 and #12) and on debugging (question #13 and #14). Such results seem to be consistent with the programming experience of the applicants. Also, errors in question #3 and question #8 are consistent with the findings on abstraction discussed in subsection 4.2.1.

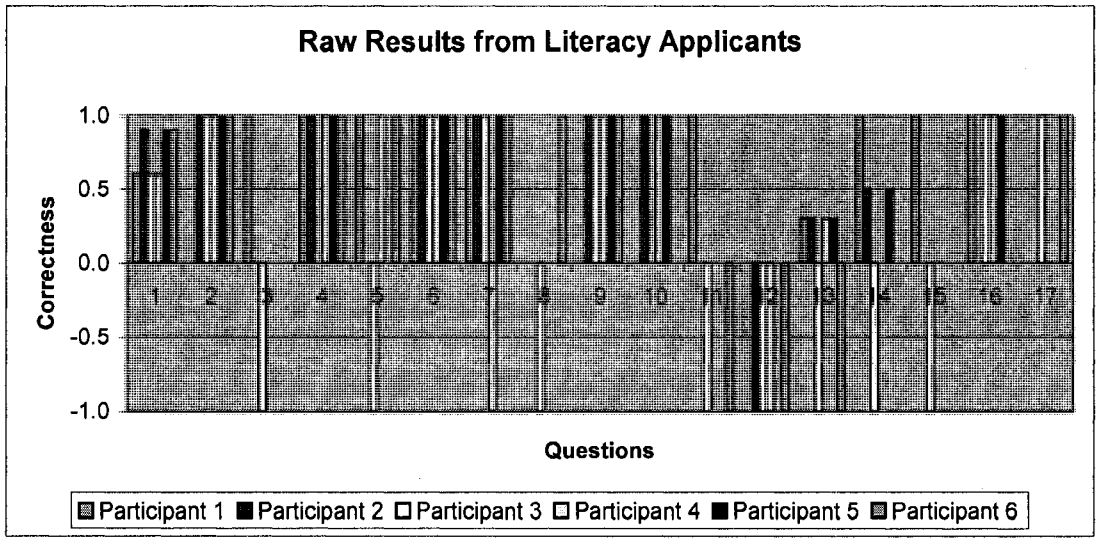

FIGURE 4.4. Results from Applicants with Programming Experience at Literacy Level. A missing bar indicates a wrong answer in the specific question where it should appear. Bars with negative value represent skipped questions. Bars with values lower than 1.0 only appear in items with sub-questions.

4.2.4 Results from the Experienced Subgroup

Figure 4.5 presents results from five participants with programming experience. Simple inspection of the chart seems to indicate that results are consistent with the programming experience of the applicants. Very few questions were skipped, and by different applicants. Difficulties with questions #3 and #8 remained consistent with the findings discussed in subsection 4.2.1.

With respect to the last question (# 17), three out of five applicants noticed that no correct answer was among the options, and therefore, they chose to skip the question and provided proper feedback about their decision. In this sense, the question proved useful to test the cognitive skills for which it was designed. (See comment in figure 4.12.)

The fact that most members from the experienced subgroup scored below 1.0 in items #1 (reading comprehension), #13, and #14 (debugging), might indicate issues with the questions themselves: inappropriate content, confusing structure or poor calibration.

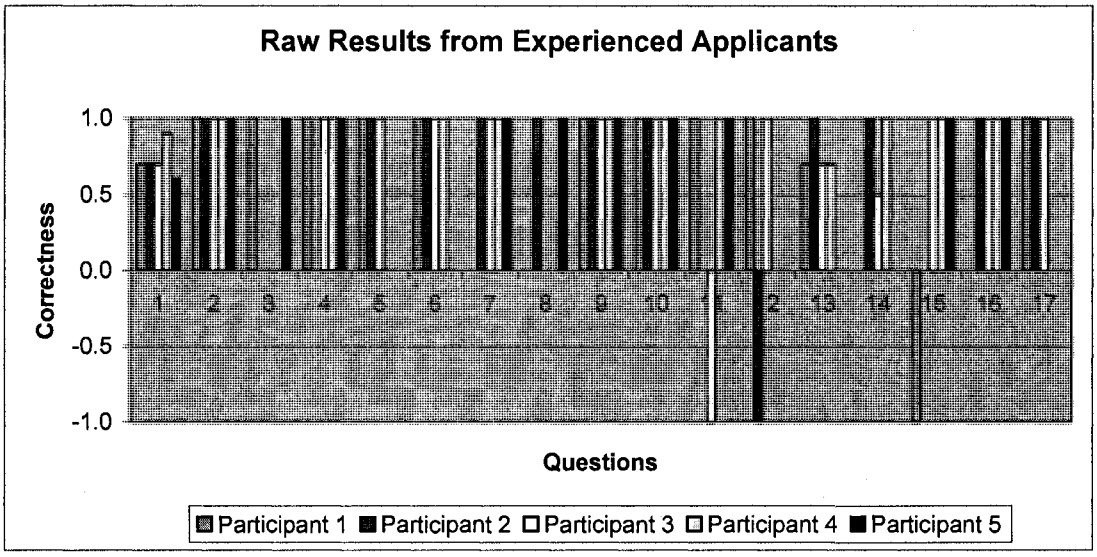

FIGURE 4.5. Results from Applicants with Programming Experience. A missing bar indicates a wrong answer in the specific question where it should appear. Bars with negative value represent skipped questions. Bars with values lower than 1.0 only appear in items with sub-questions.

4.3 The Confidence Factor

Figure 4.6 shows how confident participants felt with their answers throughout the survey. Simple inspection reveals that the literate subgroup was guessing on planning (questions #11 and #12) and debugging (question #13), while the experienced subgroup was quite sure on the answers.

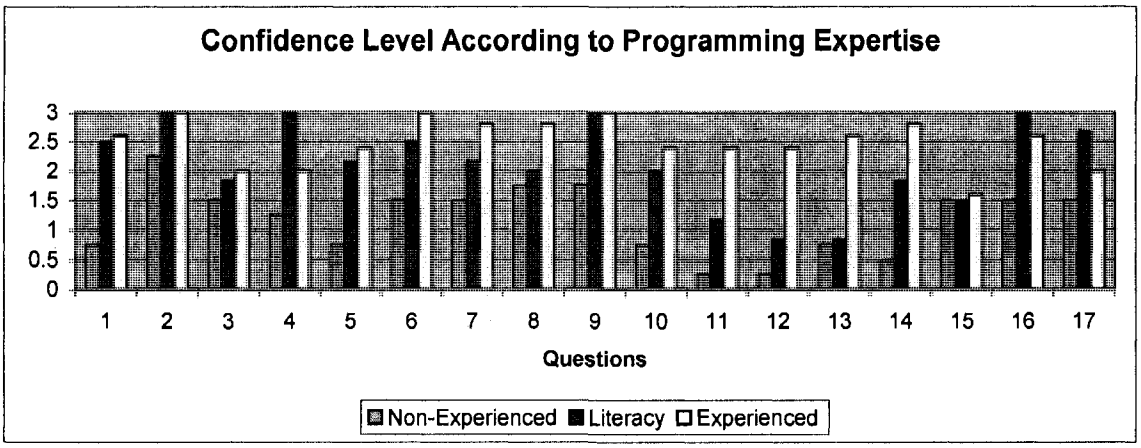

FIGURE 4.6. Answer Confidence to Level of Programming Expertise. A confidence level of 1 corresponds to guessing the answer, level 2 is for some knowledge, and level 3 is for certainty.

The confidence level was used to weight the raw scores provided by the survey according to table 3.6. For example, if the participant was guessing, the answer was disregarded because it would not reflect strength or weakness. On the contrary, if the participant had complete certainty, the value of the answer was magnified to reflect either a strong skill or a potentially serious misconception. The whole discussion on section 4.4 is based on results adjusted after confidence level.

4.4 Analysis of Skill Strength

Figures 4.7 to 4.9 show the strength of skills from the literate and experienced subgroups. The skills appear encoded in the horizontal axis (table 3.3). Simple inspection of figure 4.7 shows that the experienced subgroup had the strongest skills when tested on planning and process analysis.

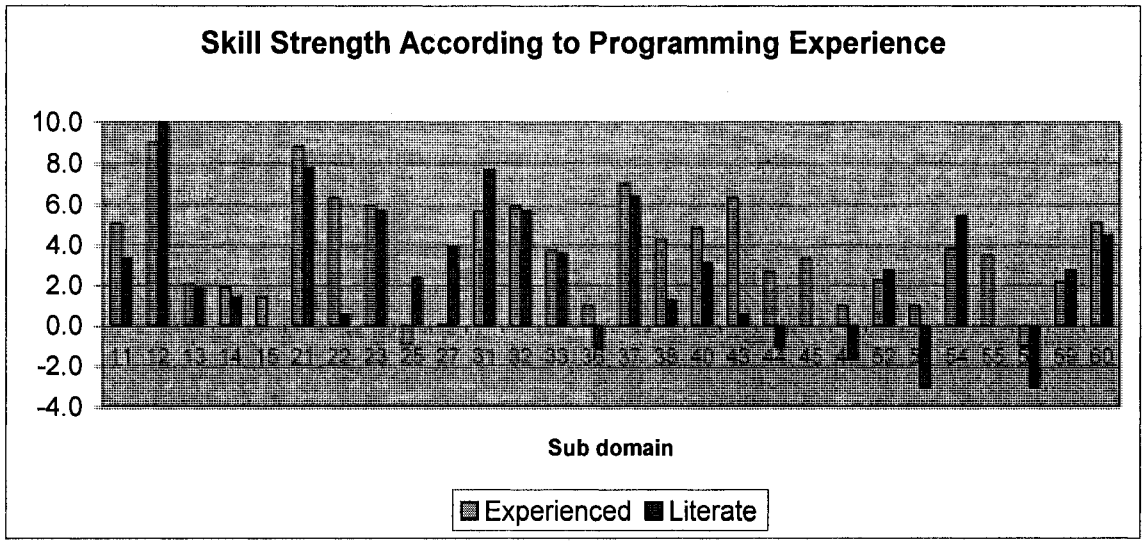

FIGURE 4.7. Average Skill Strength from TAPSS 2.0 Pilot Administration. Very good: 6-10, Good: 2-6, Base: 0-2, Needs attention: less than 0.

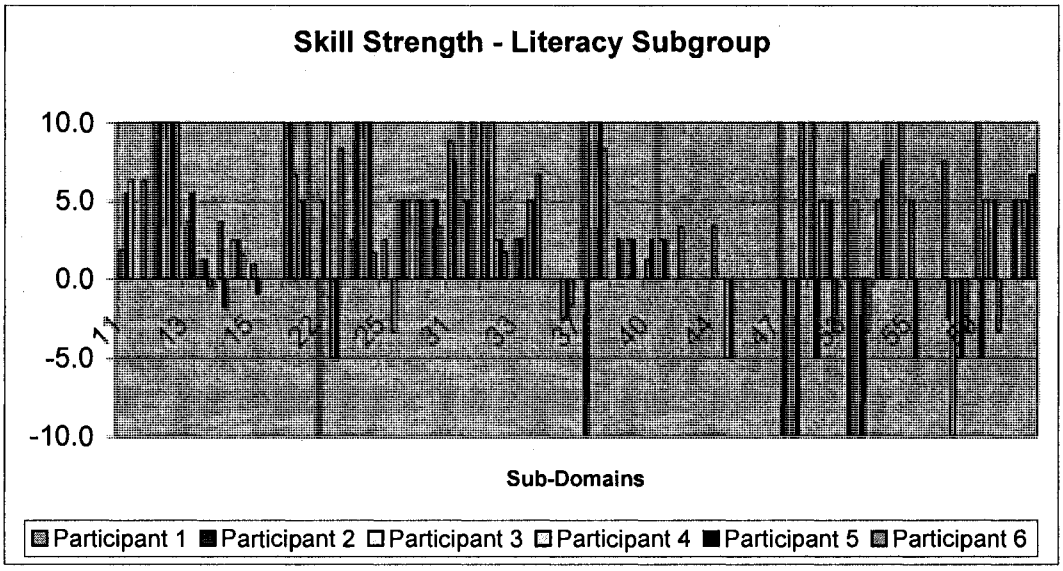

FIGURE 4.8. Skill Strength from Participants with Programming Experience at Literacy Level. Very good: 6 to 10, Good: 2 to 6, Base: 0 to 2, Attention needed: 0 to -5, Misconception: -5 to -10.

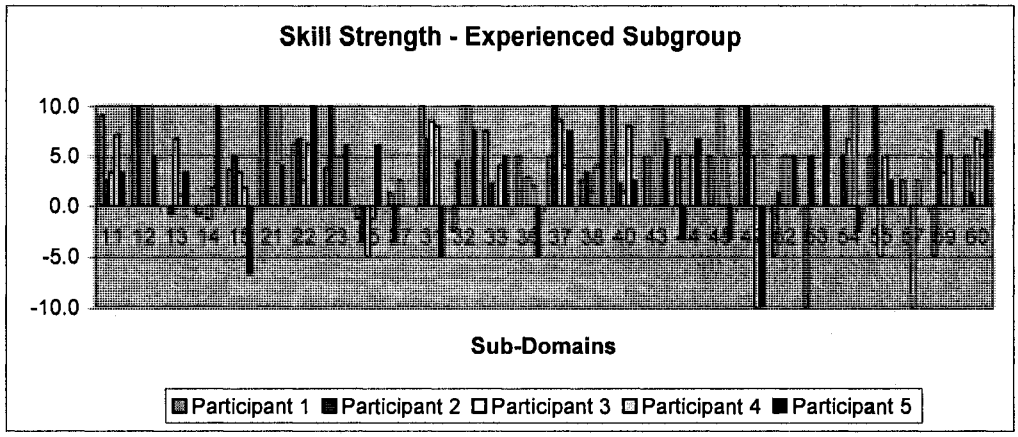

FIGURE 4.9. Skill Strength from Participants with Programming Experience at Literacy Level. Very good: 6 to 10, Good: 2 to 6, Base: 0 to 2, Attention needed: 0 to -5, Misconception: -5 to -10.

4.5 Purposely-Selected Cases

4.5.1 Looking at Individual Level

The previous charts only show aggregated data, so the information inferred may not be representative of each member in the group. We show two cases in which a closer look can be very helpful to better understanding the problem-solving abilities of the participant.

Figure 4.10 shows scores, both raw and adjusted, of an applicant from the literate subgroup. It can be seen that the applicant did well in most questions, guessed in two, and had only two minor errors. However, a major issue seems to be detected by the second question (the error appears magnified by the confidence level.)

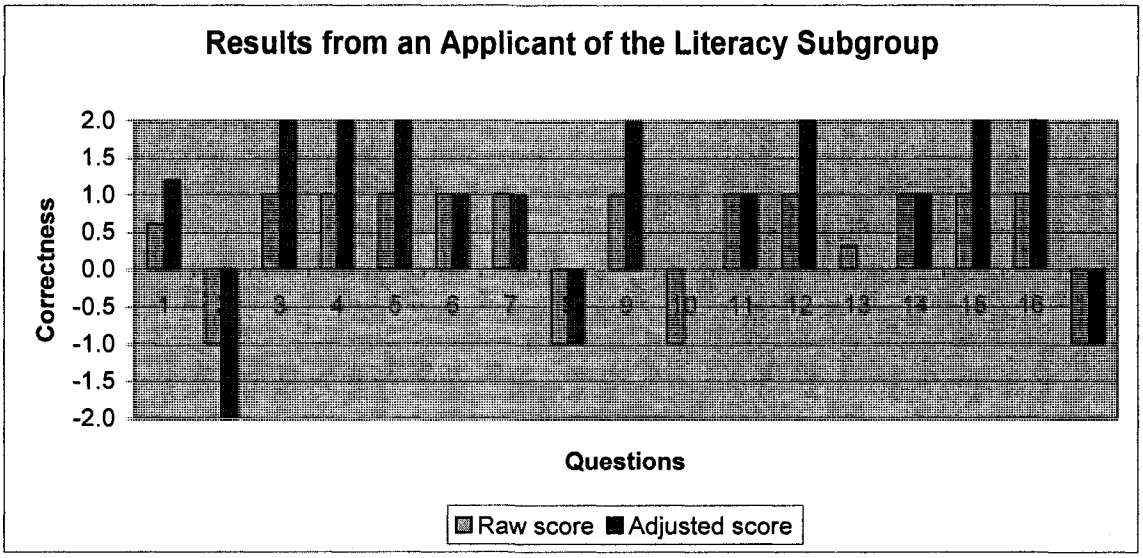

FIGURE 4.10. Results from an Applicant with Programming Experience at Literacy Level. Scores were adjusted after applicant's confidence level (second bar): Twice the height - very confident; same height - somewhat confident; no bar - just guessing.

Nevertheless, the same applicant later commented:

> *"I misread the second question, but I realized too late. It was asking the word in the row, not in the column... well... I guess that was the idea, became aware that I don't read well..."*

Such comment can be interpreted as the participant's ability to reflect on her own work and explain corrections if needed.

Another example is shown in figure 4.11. The participant skipped most questions but left several personal comments. The fact that he went through all the questions, and even had the time to write comments, could indicate motivational issues rather than fragile problem-solving skills. If that were the case, and the applicant had to course introductory programming, the motivational aspect should be addressed before instruction.

1. Skipped: Don't know the answer.
2. Error 109: Rows confused by columns.
3. Skipped: Don't know the answer.
4. Skipped: Don't know the answer.
5. Skipped: Don't know the answer.
6. Skipped: "I hate puzzles."
7. Skipped: "I am dumb."
8. Correct (guess.)
9. Correct (completely sure.)
10. Skipped: "I stink at math."
11. Skipped: "What is an integer?"
12. Skipped: "Takes too long to figure out."
13. Correct (guess.)
14. Skipped: "I don't program much."
15. Error
16. Skipped: "Lost me."
17. Skipped: "I am too lazy."

FIGURE 4.11. Comments on the Submission from a Non-Experienced Applicant.

4.5.2 General Feedback

"Hey! What was the answer to the last problem? I checked the arithmetic several times but couldn't find the option that matched. Were we supposed to skip it? I did it after noticing that this option had an explanation like 'Answer not found.'"

FIGURE 4.12. Comment on the Question Regarding Critical Thinking and Attention to Details. (Literate subgroup.)

I have to admit it wasn't easy to answer the questionnaire, but I like it.

I have some comments:

1. *Because it starts with prime numbers I felt confused. Maybe it could be good an introduction to the topics to be addressed.*
2. *Please, start with easy questions... That stuff about the primes was very difficult... maybe I'm not very acquainted...? I felt I was lacking a lot of knowledge from computing, math and algebra.*
3. *That idea of having easy questions first is to motivate the user to continue and give him the sensation that she can [solve] that question and the next one.*
4. *It says that the word "blind" is in the 4th line, but it's in the 3rd [one].*
5. *Do you take time to answer?*
6. *I didn't understand the calculator [question].*
7. *Please, remind the directions, I ended using paper and pencil for computations, was it allowed?*
8. *Problem 17 is [being] repeated.*

FIGURE 4.13. General Comment on the Survey (Literate subgroup.)

Author: *Hey! Good to find you online. Could you help me to check my program?*

Tester: *Of course, what program?*

Author: *http://www.cs.jhu.edu/~jorgev/survey.htm*

Tester: *Oh! I remember, you gave me this about one year ago, right?*

Author: *But in paper.*

Tester: *In fact, it was in a file and I returned it by e-mail.*

Tester: *It's entertaining.*

Tester: *In the 11, the calculator doesn't have the functions wr and int...*

Author: *I remove it on purpose.*

Tester: *Why?! Without visual aid I have to use the calculator, I wanted to do it mentally.*

Author: *You just need paper.*

Tester: *The procedures table is not quite clear...*

Tester: *After step 9, isn't a "=" missing?*

Tester: *More, two more columns ahead, before the end of process, in the 13, there is no instruction to display the results.*

Author: *Oops! Sorry. Any suggestion to simplify the table?*

Tester: *Rather than simplify, don't omit operations. Otherwise, you'd need to add the option "none of the above," but that is a cheap trick.*

Tester: *The writing in the 15 is funny. At first glance I understand that you move up in the first column, and down in the second, but that does not allow to make a diagonal.*

Author: *Came from and old test I updated. However, there is no diagonal.*

Tester: *I skipped, I didn't understand it.*

Tester: *I'm on the 17 now... No option seems good...*

Author: *You are right.*

FIGURE 4.14. Survey Review through and Online Interview. (Experienced programmer.)

Chapter 5

Conclusions and Impact

5.1 Overview

In this dissertation we have addressed the problem of understanding what prevents many college students from learning computer programming. Our main assumption has been that behind this situation there are factors indirectly related with coursework like deficiencies in reading, arithmetic, or algebra; abilities which mastery precedes the programming level. To this end, we have modeled and developed a test specialized in algorithmic problem-solving skills, aiming to survey fundamental abilities in computer programming.

The test is comprised of three main elements: a questionnaire, a surveying process, and a scoring model. Each item in the questionnaire corresponds to a problem-solving skill fundamental for programming (see table 5.1.) Most questions are multiple-choice with multiple valid answers and span several sub-ability domains (see subsection 3.5.6). The surveying process involves requesting the applicants to rate their confidence on their own answers, and gives them the possibility to skip questions. The scoring model consists of weighting answers

according to the corresponding confidence level, and updating several skill-tracking variables according to the correctness of each answer.

The test was implemented as a web-based application dubbed TAPSS 2.0, and tested during the period October 25 to November 19, 2007. Comments and reflections on the results and their impact are discussed in the following sections.

Question No.	*Skill*
1	Reading comprehension: passage level
2	Reading comprehension: word level
3	Problem identification
4	Reverse word problem translation
5	Word problem translation
6	Word problem (logic) solution
7	Word problem (arithmetic) solution
8	Following (algebraic) procedure
9	Boolean logic: inequalities
10	Boolean logic: truth tables
11	Stepwise planning: symbolic sequence
12	Stepwise planning: statement sequence
13	Debugging pseudocode: finding errors
14	Debugging pseudocode: correcting errors
15	Following instructions
16	Following procedure
17	Critical thinking & details

TABLE 5.1. Questionnaire Items and Corresponding Skills.

5.2 Discussion of Key Findings

5.2.1 The Questionnaire

Our preliminary questionnaire was developed in informal way, close in content and form to classwork exercises and exams. Thus, its effectiveness as skill testing instrument was unclear. In contrast, the careful design of the final questionnaire has allowed reviewing each question. Table 5.2 summarizes our review of each item in the survey.

- **Abstraction**

Although a low level of correctness might be due to an unsuitable question, it can result from a highly skill-specific one. Such seems to be the case of question #3, an information abstraction problem, and question #8, a following-procedure problem with abstract answer.

Results on question #3 can be interpreted as confirmation of the McCracken's group observation "the most difficult part for students seemed to be abstracting the problem to be solved from the exercise description." [117, p.133] With respect to question #8 (fig. 3.10), a problem the author has repeatedly used when teaching introductory programming for pre-testing and discussion. In our experience, most students give a numeric answer, and only very few students —not necessarily experienced in programming— provide an algebraic one.

Because both questions deal with abstraction, in its two main meanings — detail removal and non-concrete thinking, the results can be aligned with Kramer's observations on the problem of abstraction in computer education [100].

Furthermore, these two questions can serve as a model to create a specific instrument to test abstract thinking and abstraction skills in computer science students. (Kramer's research has been unable to find a suitable instrument for such purpose, [100, p.42].)

- **Critical Thinking and Details**

Item #17 was a multiple-choice question with no correct answer provided, and it was aimed to test attention to details (i.e., noticing lack of correct answer) and critical thinking skills (i.e., how to proceed.) From a problem-solving perspective, a student that knows the correct answer, and decides to skip this question, makes a better decision- than someone that knows the answer but selects any of the provided ones.

From a programming perspective, this kind of question reflects attention to details and awareness of the reasons behind actions, both important qualities for making programs. For example, if some part of the solution process is unknown, it can be better to document the issue and skip it, than become stuck on it. In a different case, not taking the options at face value can be helpful during the initial testing of the program, particularly because a number of students tend to accept results from incorrect programs just because they appeared in a book, were given by the instructor or, simply, were executed by a computer.

From the results presented in the previous chapter, it is clear that people with the experienced subgroup had the advantage over the other two. Also, the question worked as expected (see comments on figures 4.12 and 4.14.)

- **Reading**

Regarding question #1, general reading comprehension, despite the fact that all the answers can be found in the passage, the topic seemed to deeply affect the attitude of the student towards the question, and therefore, his/her confidence and answers. The situation was particularly noticeable on the reading about prime numbers. A couple of examples: One applicant made a specific remark about how difficult was this question (see fig. 4.12), and other who did well in a reading no related to prime numbers, failed when asked to find the best phrase summarizing the prime numbers reading (question #3).

Because correctness of this question was lower than expected, its effectiveness is unclear and a thorough review of this item will follow.

- **Planning and Debugging**

The fact that most members from the subgroup with programming experience scored below 1.0 in problems regarding stepwise planning (questions #11 and #12) debugging pseudocode (questions #13 and #14), might indicate issues with the questions themselves: inappropriate content, confusing structure or poor calibration. At this point, their effectiveness is unsatisfactory and a thorough review of these items will follow.

Question	Effectiveness	Comment
General reading (#1)	Unclear	The question worked as expected, but its contribution to the results is unclear. Either the readings or the sub-questions might not be suitable for the goals of the test.
Specific reading (#2)	Very good	The short reading and the question were adequate. The question properly detected the common confusion between rows and columns.
Summary (#3)	Unclear	Most applicants had problems to find the best phrase summarizing the reading. It is related to question #1, so the reading topic might be the cause.
Reverse word problem (#4)	Very good	The attention to detail in reading both, question and options, worked as expected. (See discussion in subsection 3.4.2)
Arithmetic (#5, #6, #7)	Very good	Worked as expected to appreciate basic arithmetic skills.
Following procedure with algebraic answer (#8)	Unclear	Most applicants give a numeric answer rather than an algebraic one. This indicates a problem either with the question or the intended skill. (See discussion in subsection 4.2.1)
Inequalities (#9)	Good	In general and regardless of programming experience, the question was answered correctly. However, the question might be too simple.
Logic propositions (#10)	Good	The question seemed to be problematic for people with no programming experienced at all. However, the question might be too simple.

TABLE 5.2 Main Comments about Questionnaire Items.

Question	Effectiveness	Comment
Planning (#11, #12)	Unsatisfactory	The results were poor even in the experienced subgroup. Question, options or notation could have complicated this problem.
Tracing pseudocode (#13, #14)	Unsatisfactory	In general, only the experienced subgroup was able to solve these problems. Although this result is not surprising, the question format might have prevented other people from answering.
Procedure comprehension (#15, #16)	Very good	Both questions seemed to be useful to appreciate this skill. Simpler problems would help to better appreciate the skills of the non-experienced subgroup.
Attention to detail and critical thinking (#17)	Very good	The question worked as expected, applicant needed to observe that no viable answer was provided and that skipping the option was a valid course of action.

TABLE 5.2 Main Comments about Questionnaire Items.

5.2.2 Self-Rating Confidence

Although a couple of participants commented that self-rating was a tiresome process, we think the effort involved mimics valuable attributes required by the programming activity:

- **Perseverance**. Oftentimes, patience and perseverance are key attributes for succeeding in every stage of the programming lifecycle (section 2.5) to deliver a working program. An applicant who gets tired of going through all aspects of the surveying process, or quits it, might have a hard time if enrolled in a programming course. In this context, quitting is clearer indication of a major

learning hazard: programming demands tenacity and ability to persevere, attributes also expressed by students in ChMura's study [37, p.56a].

- **Self-reflection**. Despite success in making a working program, truly understanding requires reflection and certainty of every aspect of the problem solving and programming processes involved [74]. In some way, the survey promotes this when asking the participant to rate the confidence in his/her answer.
- **Cautious confidence**. Constant doubt is not helpful in making programs, but overconfidence is not useful either (a programmer who overestimates his/her abilities easily oversees pitfalls.) The surveying process implemented can potentially detect overconfidence by presenting the same problem in a different context, or with different answers. In addition, it can help to prevent it by promoting double-checking through the self-rating confidence mechanism.

5.2.3 Motivation

Although the results from the non-experienced subgroup were lower than expected, the situation is not uncommon in programming classes. Students without any computing experience, non-science/engineering majors, or simply non-interested, often perform poorer than expected.

In the case of the non-experienced subgroup, the performance might be attributed to lack of attention to the test, a situation that can occur without proper motivation, like a grade, questions being difficult or beyond their interests.

Currently, TAPSS 2.0 does not provide enough information regarding the skills of applicants with characteristics resembling those of the non-experienced sub-group. A student performing at this level would benefit of thorough academic advice if planning to attend programming courses.

5.2.4 Flexible Surveying Mechanism

After our experience in fine-tuning TAPSS 2.0 test engine, by analyzing the data provided, further substantiated the data with interviews, it was clear that the software architecture was flexible enough to allow easy maintenance and upgrade.

At this point in time, the system is ready to be enhance by incorporating hints if an almost-correct option is selected, provide immediate feedback after submitting an answer, and allow chained questions (i.e., one problem broken down into several questions.)

5.3 Contributions

5.3.1 Theoretical Aspects

- **New Items to Test Thinking.**

As pointed by Mayer [114], Kramer [100], and Hazzan (commented in [100, p. 42]) there is a need for specific instrument to test logic, abstract thinking and abstraction skills in computer science students. To this end, our questions involving abstract thinking (see table 5.1, questions #3 and #8) can serve as model to design such kind of instruments.

- **Qualitative Research**

Our study has followed a qualitative approach to identify problem-solving abilities and track student progress. Nevertheless, it has also set the foundation to formalize a methodology, as well as an effective instrument, to gather statistics on basic problem-solving skills.

In addition, the methods and processes followed in our study can help programming instructors to create their own tools and questions, as well as to promote and guide further studies.

- **Identification of Problem-Solving Skills**

Throughout our study we have been identifying a minimum of skills and elements to test with the surveying instrument. Besides, the technique implemented allows creating profiles based on development of problem-solving skills, which in turn, can be used to perform longitudinal studies (i.e., track student progress throughout a

course) and add an outcomes-based assessment component to computer programming courses.

In addition, we have set several elements to monitor and track during the surveying process: (*i*) A model of five ability domains involved in problem solving (5-AD), (*ii*) an inventory of problem-solving skills related to introductory programming, and (*iii*) an inventory of common programming pitfalls.

5.3.2 Practical Benefits

Besides providing a pre-testing tool the author will use in future programming courses, this study will help to set guidelines in three instructional directions: (*i*) constructivist-grounded training for teaching assistants specialized in computer programming, (*ii*) curricula assessment for introductory programming courses, and (iii) provide students with more effective feedback regarding computer programming skills to become fluent with information technologies.

5.4 Future Work

The preceding chapters have discussed the development of a test of algorithmic problem-solving skills, its web-based implementation, and the results from its pilot administration. Based on the analysis of results and feedback received, several questions have to be revisited, and the questionnaire as whole refined.

The items on stepwise planning and pseudocode debugging require particular attention. Interviews with colleagues and students will be conducted to better appreciate how useful is the current format of such items and how improve them. In

parallel, the surveying process will be enhanced to enable study of quantitative aspects of problem solving skills.

Additionally, our preliminary results seem to indicate that this system can be converted into an instrument for developmental assessment if administered several times through the course. We would results obtained during the first administration as baseline and then proceed to apply longitudinal tracking. Furthermore, TAPSS 2.0 could serve as platform to build an adaptive tutorial to help students in developing skills in language-independent algorithmic thinking, by detecting learning issues and critical skills, and taking the appropriate path to help them reinforce the learning process of algorithm creation. At this point in time, the system is ready to be enhanced by incorporating hints if an almost-correct option is selected, provide immediate feedback after submitting an answer, allow chained questions (i.e., one problem broken down into several questions) and follow Cabral-Vasconcelos heuristics for problem-solving [31, 166]. (A problem-solving strategy modeled as a sequence of stages to be performed conditionally. At each condition point there is a questionnaire to help students in the thinking process, promote review of their work at each stage, find necessary concepts, independent and dependent variables, formulas, predict and contrast results after following the designed strategy, and correct them if needed.)

Furthermore, innovative training methods and tools can be explored because of the flexibility of the test engine and the question bank. For example, the surveying process could be tailored to practice web-search skills like finding and understanding information from the Wikipedia[20]

[20] Suggested by G. Masson on December 18, 2007.

Other surveying aspects still requiring further study are timing and guessing: How important is tracking time for each questionnaire item, or how random is a guess. Such study may involve psychological factors that will require advice from experts on that field.

5.5 Summary

This dissertation has presented the development of a test for algorithmic problem-solving skills aiming to survey abilities fundamental to computer programming. The work has been grounded in a constructivist theoretical framework and has followed a hermeneutic approach to understand and integrate common programming errors, programming-specific thinking styles and problem-solving ability domains. The web-based survey prototype, along with the methodological framework associated, are aimed to provide programming instructors with information and resources to differentiate instruction according to diverse levels of problem-solving abilities, as well as help students to reflect on their problem solving strengths, while gaining a deeper understanding of the knowledge, abilities, and cognitive processes needed to become skillful in creating algorithms and elaborate computer programs.

Bibliography

[1] ACM and IEEE Computer Society. *Computing Curricula 2001*, Computer Science. December 2001.

[2] Adair, J., *Problem Solving, A Top-Down Approach.* Scott, Foresman and Co. Glenview, Il. 1989.

[3] Aiken, R., et al, "Fluency in Information Technology: A Second Course for Non-CIS Majors." *Proceedings of the SIGCSE 2000 Symposium.* Austin, TX. March 2000. pp.280-284.

[4] Al-A'ali, M., "Evaluation of Teaching an Arabic Programming Language (ARABLANG)." *Computers in Education Journal.* April-June, 2007. pp.37-49.

[5] Ala-Mutka, K., "Problems in Learning and Teaching Programming." *Technical Report for the Codewitz Project.* Tampere University of Technology Institute of Software Systems. (Circa 2004). <http://www.cs.tut.fi/~codewitz> Accessed on December 5, 2006.

[6] Allison, I., et al., "A Virtual Learning Environment for Introductory Programming." *Proceedings of the 3rd Annual Conference of the LTSN Centre for Information and Computer Sciences,* Loughborough, UK. August 2002. pp.48-52.

[7] Alvesson, M., and Skolberg, K., *Reflexive Methodology*. SAGE publications. London, 2000.

[8] Armour, P. G., "The unconscious Art of Testing." *Communications of the ACM.* January 2005. pp.15-18.

[9] Bailey, J. L. and Stefaniak, G., "Industry Perceptions of the Knowledge, Skills, and Abilities Needed by Computer Programmers." *Proceedings of the 2001 ACM SIGCPR.* April 2001. pp.93-99.

[10] Barnes, D. J., "Introductory Problem Solving in Computer Science." *5th Annual Conference on the Teaching of Computing,* Centre for Teaching Computing, Dublin City University, Dublin 9, Ireland. August 1997. pp.36-39.

[11] Barnes, D. J., "Students Asking Questions: Facilitating Questioning Aids Understanding and Enhances Software Engineering Skills." *Inroads, The SIGCSE Bulletin,* December 1997. pp.38-41.

[12] Barnes, D. J., "Public Forum Help Seeking: the Impact of Providing Anonymity on Student Help Seeking Behaviour." *Proceedings of the Computer Based Learning in Science Conference.* Enschede, the Netherlands, July, 1999.

[13] Bayman, P. and Mayer, R. E., "A Diagnosis of Beginning Programmers' Misconceptions of BASIC Programming Statements." *Communications of the ACM.* September, 1983. pp.677-679.

[14] Beaubouef, T., "Why Computer Science Students Need Language." *Inroads, The SIGCSE Bulletin,* December 2002. pp.57-59.

[15] Beaubouef, T., "Why Computer Science Students Need Math." *Inroads, The SIGCSE Bulletin,* December 2003. pp.51-54.

[16] Begum, M., "An Ontology for Teaching Programming." *Doctoral Consortium. SIGCSE 2004 Symposium*. ACM. Norfolk, VA. March 2004.

[17] Beisy, C. and Mayers, M., "What IT Labor Shortage." *Ubiquity, ACM IT Magazine*. February 21, 2000.
< http://www.acm.org/ubiquity/views/c_beise_1.html> Last accessed: June 18, 2007.

[18] Bell, D., et al., *The Essence of Program Design*. Prentice Hall. London, UK. 1997.

[19] Ben-Ari, M., "Constructivism in Computer Science Education." *Proceedings of the SIGCSE 1998 Symposium*. ACM, Atlanta, GA. February 1998. pp.257-268.

[20] Ben-Ari, M., "Constructivism in Computer Science Education." *Journal of Computers in Mathematics and Science Teaching*. Vol.20, No.1, 2001. pp.45-73. (AACE digital library: http://dl.aace.org/6381)

[21] Ben-Ari, M., et al. "Computer Architecture and Mental Models", *Proceedings of the SIGCSE 2005 Symposium*. ACM, St. Louis, MI. February 2005. pp.101-105.

[22] Berlinsky, D., *The advent of the Algorithm, the Idea that Rules the World*. Harcourt, Inc. New York, NY. 2000.

[23] Bonar, J. and Soloway, E., "Preprogramming Knowledge: A Major Source of Misconceptions in Novice Programmers." In: Soloway & Spohrer, *Studying the Novice Programmer*, 1989. pp.325-354.

[24] Bores, R. And Rosales, R. *Computación: Metodología, Lógica Computacional y Programación*. McGraw Hill, México. 1994.

[25] Bruce, K. B., "Controversy on How to Teach CS1," *Inroads, The SIGCSE Bulletin*, December 2004. pp.29-34.

[26] Bruckman, A. and Edwards, E., "Show we Leverage Natural-Language Knowledge? An Analysis of User Errors in a Natural-Language-Style Programming Language." *Proceedings of CHI'99*. Pittsburgh, PA. May 1999. pp.207-214.

[27] Bruhn R. E. and Burton, P. J., "An Approach to Teaching Java Using Computers." *Inroads, The SIGCSE Bulletin*, December 2003. pp.94-99.

[28] Buck, D and Stucki, D. J., "Design Early Considered Harmful: Graduated Exposure to Complexity and Structure Based on Levels of Cognitive Development." *Proceedings of SIGCSE 2000 Symposium*. March 2000, pp.75-79.

[29] Buck, D and Stucki, D. J., "JKarelRobot: A Case Study in Supporting Levels of Cognitive Development in the Computer Science Curriculum." *Proceedings of SIGCSE 2001 Symposium*. February 2001, pp.16-20.

[30] Burton, P. J., and Bruhn R. E., "Teaching Programming in the OOP Era." *Inroads, The SIGCSE Bulletin*, June 2003. pp.111-114.

[31] Cabral, L., et al., "Problem Solving." *Contactos*, Vol.II, No.8, Universidad Autónoma Metropolitana, México, 1995. pp.42-51,

[32] Carlisle, M. C., "RAPTOR: Introducing Programming to Non-Majors with Flowcharts." *Journal of Computing Sciences in Colleges*, Volume 19 Issue 4, April 2004.

[33] Carlisle, M. C. "RAPTOR: a Visual Programming Environment for Teaching Algorithmic Problem Solving," *Proceedings of the SIGCSE 2005 Symposium.* ACM, St. Louis, MI. February 2005.

[34] Carroll, J. M., *The Nurnberg Funnel: Designing Minimalist Instruction for Practical Computer Skill*, MIT Press. Cambridge, MA. 1990.

[35] Centro de didáctica UNAM. *Manual de Didáctica General.* Programa Nacional de Formación de Profesores. Asociación Nacional de Universidades e Institutos de Enseñanza Superior. México, 1972.

[36] Chai, I., "Pedagogical Framework Documentation: A Research Proposal." *Empirical Studies of Programmers '97.* Student Workshop, Alexandria, Virginia, 24-26 October, 1997

[37] ChMura, G. A., "What Abilities are Necessary for Success in Computer Science?" *Inroads, The SIGCSE Bulletin*, December 1998. pp.55a-58a.

[38] Clear, T., "The Waterfall is Dead – Long Live the Waterfall!!" *Inroads, The SIGCSE Bulletin*, December 2003. pp.13-14.

[39] Colwell, B., "Engineers, Programmers, and Black Boxes." *IEEE Computer Magazine.* March 2005. pp.8-11.

[40] Coombs, M. J. and Hartley, R. T., "Debugging User Conceptions of Interpretation Processes." *Proceedings of the 5th National Conference on Artificial Intelligence.* Philadelphia, PA, August 11-15, 1986. pp.303-307

[41] Cooper, S., et al, "Developing Algorithmic Thinking With Alice." *Proceedings ISECON 2000,* Philadelphia, PA, 2000. pp.506-539.

[42] Cunningham, S., "Graphical Problem Solving and Visual Communication in the Beginning Computer Graphics Course." *Proceedings of the SIGCSE 2002 Symposium.* Covington, KY. February-March 2002. pp.181-185.

[43] Crawford, C., "How to Think: Algorithmic Thinking." *Journal of Computer Game Design* v.7, 1993. http://www.erasmatazz.com/library/JCGD_Volume_7/Algorithmic_Thinking.html (last accessed: August 21, 2003.)

[44] Crotty, M., *The Foundations of Social Research: Meaning and Perspective in the research Process.* SAGE Publications, London 1998.

[45] Curtis A. C., et al., "A Methodology for Active, Student-Controlled Learning: Motivating our Weakest Students." *Proceedings of the SIGCSE 1996 Symposium.* pp.195-199.

[46] Dale, N. and Lewis, J., *Computer Science Illuminated.* Jones and Bartlett Publishers.Sudbury, MA. 2004.

[47] Decker, A., "Evaluating Skills Necessary for Computer Science." *SIGCSE 2003 Doctoral Consortium.* 2003.

[48] Decker, A., "How Students Measure Up: Creation of an Assessment Tool for CS1." *SIGCSE 2004 Doctoral Consortium.* 2004.

[49] Decker, A. *How Students Measure Up: Assessment Instrument for Introductory Computer Science.* Ph.D. Dissertation. University at Buffalo. May 2007.

[50] Deek, F.P., et al. "A common model for problem solving and program development." *IEEE Transactions on Education.* November 1999. pp.331-336.

[51] Deek, F.P. and McHugh, J. A. "Problem Solving and Cognitive Foundations for Program Development: An Integrated Model." *Sixth International Conference on Computer Based Learning in Science.* Nicosia, Cyprus, 2003. pp. 266-271.

[52] Deimel Jr., L. E. "The Uses of Program Reading." *Inroads, The SIGCSE Bulletin*, June 1985, pp.5-14.

[53] DeFranco-Tommarello, J., *Literature Review of Collaborative Problem Solving and Groupware in the Software Development Domain.* State of the Art Review Paper, New Jersey Institute of Technology. (circa 2002). <http:// web.njit.edu/~hiltz/SOTA_JDT.doc>, retrieved on November 23, 2007.

[54] de Lemos, M. A., and Barros, L. N., "A Didactic Interface in a Programming Tutor." Proceedings of the 11th International Conference of Artificial Intelligence in Education. Sydney, Australia. July 2003. pp.433-440.

[55] Denning, P. J. "Mastering the Mess." *Communications of the ACM*, April 2007. pp.21-25.

[56] DePasquale, P. J. *Improving Learning of Programming Languages Through the Implementation of Levels in Program Development Environments.* Ph.D. Dissertation. Virginia Polytechnic Institute and State University. Summer 2003.

[57] desJardin, M. et al., "Representing a Student's Learning States and Transitions". *AAAI Technical Report*, 1995.

[58] Dromey, R. G. *How to Solve it by Computer.* Prentice Hall. Englewood Cliffs, NJ. 1982.

[59] Du Bolay, "Some Difficulties of Learning Programming." In: Soloway & Spohrer, *Studying the Novice Programmer*, 1989. pp.283-300.

[60] Egan, M. L., "Students with Asperger's Syndrome in the CS Classroom." *Proceedings of the SIGCSE 2005 Symposium.* ACM, St. Louis, MI. February 2005. pp.27-30.

[61] Elsehoff, J. L. and Marcotty, M., "Improving Computer Program Readability to Aid Modification." *Communications of the ACM,* August 1982. pp.512-521.

[62] English, J., "Experience with a Computer-Assisted Formal Programming Examination." *Proceedings of ITiCSE'02.* p.51-54. Denmark, 2002.

[63] Etter, Delores M., *C++ for Engineers and Scientists.* Prentice Hall. 1997.

[64] Fedje, C. G., "Program Misconceptions: Breaking the Patterns of Thinking." Journal of Family and Consumer Sciences Education. Vol. 17. No. 2. 1999. pp.11-19.

[65] Fincher, S., et al., "Multi-Institutional, Multi-National Studies in CSEd Research: Some design considerations and trade-offs." *Proceedings of the 1st International Computing Education Research Workshop,* ICER 2005

[66] Fitzgerald, S., et al., "Strategies that Students Use to Trace Code: An Analysis Based in Grounded Theory." *Proceedings of ICER 2005,* pp.69-802.

[67] Flowers, T., et al., "Empowering Students and Building Confidence in Novice Programmers through Gauntlet." *Proceedings of the 34th ASEE/IEEE Frontiers in Education Conference.* Savannah, GA. October 2004.

[68] Fone, W., "Improving feedback from Multiple Choice Tests." *Proceedings of ICALT'02,* Denmark, 2002. p.96.

[69] Fone, W., "A Topology and Framework to Aid the Design of Automated assessment." *Proceedings of ICALT 2003.*

[70] Fone, W., "Design of MCQ Test." *Proceeding of ITiCSE'04.* UK 2004. p.250.

[71] Gay, L. R., *Educational Research: Competencies for Analysis and Application.* Charless Merrill Publishin Co. Columbus, OH. 1976.

[72] Gibson, J. P. and O'Kelly, J., "Software Engineering as a Model of Understanding for Learning and Problem Solving." *Proceedings of the 1st International Computing Education Research Workshop, ICER 2005.* Seattle, Washington. October, 2005. pp.87-97.

[73] Gibbs, D. C., "The Effect of a Constructivist Learning Environment for Field-Dependent/Independent Students on Achievement in Introductory Computer Programming." *Proceedings of the SIGCSE 2000 Symposium.* Austin, TX. March 2000. pp.207-211.

[74] Ginat, D., "Metacognitive Awareness Utilized for Learning Control Elements in Algorithmic Problem Solving." *Proceeding of ITiCSE'01.* Canterbury, UK 2001. pp.81-84.

[75] Ginat, D., "On Varying Perspectives of Problem Decomposition." *Proceedings of the SIGCSE 2002 Symposium.* Covington, KY. February-March 2002. pp.331-335.

[76] González-Sandoval, N. *Personal Communications.* 1995-2005.

[77] Gray, S., et al., "Suggestions for Graduated Exposure to Programming Concepts Using Fading Worked Examples." *Proceedings of the International Computing Education Research Workshop,* ICER 2007. Atlanta, Georgia. September, 2007. pp.99-110.

[78] Green, T. R. G., "Instructions and Descriptions: Some Cognitive Aspects of Programming and Similar Activities." *ACM AVI 2000.* Palermo, Italy, 2000.

[79] Gross, P. and Power, K., "Evaluating Assessments of Novice Programming Environments." In *Proceedings of ICER 2005*, p.99.

[80] Guggerty, L. and Olson, G., "Debugging by Skilled and Novice Programmers," *CHI'86 Proceedings*. pp.171-174. April, 1986.

[81] Guzdial, M. and Soloway, E., "Log on Education: Teaching the Nintendo Generation to Program." *Communications of the ACM*, April 2002. pp.17-21.

[82] Guzdial, M. and Tew, A. E., "Imagineering Inauthentic Legitimate Peripheral Participation: An Instructional Design Approach for Motivating Computing Education." *Proceedings of the Second International Computing Education Research Workshop*, Canterbury, UK. 2006. pp.51-58.

[83] Haberman, B., et al., "Action Research as a Tool for Promoting Teacher Awareness of Students' Conceptual Understanding." Proceedings of *ITiCSE'03 Conference*. Thessaloniki, Greece, June 2003. pp.144-148.

[84] Haladyna, T. M., *Developing and Validating Multiple-Choice Test Items*. Lawrence Earlbaum Associates. Hillsdale, NJ 1994.

[85] Haladyna, T. M., *Writing Test Items to Evaluate Higher Order Thinking*. Allyn and Bacon. Boston, MA 1997.

[86] Henderson, P. B., "Modern Introductory Computer Science." *Proceedings of the SIGCSE 1987 Symposium*. St. Louis, MI. February 1987. pp.183-190.

[87] Hernández-Valdelamar, J. *Personal Communications*. 1998-2005.

[88] Houlahan, J. *Personal Communications*. 2000-2005.

[89] Jadud, M. C., "A First Look at Novice Compilation Behavior Using BlueJ." *16th Workshop of the Psychology of Programming Interest Group*. Conzenza, Italy. April, 2004. pp.181-192.

[90] Jenkins, T., "The Motivation of Students of Programming." *ITiCSE'01 Conference.* ACM. Canterbury, UK, June 2001. pp.53-56.

[91] Jenkins, T., "Teaching Programming – A Journey from Teacher to Motivator." *Proceedings of the 2nd Annual Conference of the LTSN Centre for Information and Computer Sciences,* Loughborough, UK. August 2001. pp.48-52.

[92] Johns Hopkins University, "600.106 (E) Pre-Programming: Algorithmic Thinking." *JHU Catalog 2005-2007.* http://catalog.jhu.edu/ (Last accessed 10/4/06.)

[93] Kelleher, C., et al., "Lowering the Barriers to Programming: A Taxonomy of Programming Environments and Languages for Novice Programmers." *ACM Computing Surveys,* June 2005.

[94] Kirov, N. "University education on programming." *International Seminar,* New Bulgarian University. Borovec, September 25-26, 2004. http://www.math.bas.bg/~nkirov/2004/borovec.html (Last accessed 10/4/06.)

[95] Knuth, D. E., "Algorithmic Thinking and Mathematical Thinking." *American Mathematical Monthly.* No. 92. 1985. pp.170-181.

[96] Knuth, D. E., *Selected Papers on Computer Science.* Cambridge University Press. Cambridge, MA. 1996.

[97] Kolikant, Y. B-D., "Students' Alternative Standards for Correctness." *Proceedings of ICER 2005,* p.37.

[98] Koppelman, H., "Teaching Abstraction Explicitly." Proceedings of the *ITiCSE'01 Conference,* Canterbury, UK. ACM. June 2001. p.191.

[99] Kramer, D., "Algorithms should Mind your Business." *OffshoreDev.com*, July 23, 2002. http://www.outsourcing-russia.com/docs/?doc=680 (Last accessed 9/9/05.)

[100] Kramer, J., "Is Abstraction the Key to Computing?" *Communications of the ACM*, April 2007. pp.36-42.

[101] Krishna, A. K. and Kumar, A. N., "A problem generator to learn expression: evaluation in CSI, and its effectiveness." *Journal of Computing Sciences in Colleges.* April 2001. pp.34-46.

[102] Krishna, R., "Databases and Artificial Intelligence: Infusing Critical Thinking Skills into Content of AI Course." *ITiCSE'05 Conference,* Portugal. ACM. June 2005. pp.173-177.

[103] Koltun, P., et al, "Progress Report on the Study of Program Reading." ACM publications, 1983. pp.168-176.

[104] Kummerfeld, S. K. and Kay, J., "The Neglected Battle Fields of Syntax Errors." Australasian Computing Education Conference, ACE 2003. Adelaide, Australia.

[105] Lahtien, E., et al., "A Study of the Difficulties of Novice Programmers." *ITiCSE'05 Conference,* Portugal. ACM. June 2005. pp.14-19.

[106] Lane, H. C., "A preventing Tutoring System for Beginning Programming", *SIGCSE 2004 Doctoral Consortium.* 2004.

[107] Lewis, T. L., "Using Individual Differences to Teach to Introductory Object-Oriented Design Students." *SIGCSE 2004, Doctoral Consortium.* 2004.

[108] Linn, M. C. and Clancy, M. J., "The Case for Case Studies of Programming Problems." *Communications of the ACM.* March 1992. pp.121-132.

[109] Lister, R. and Leaney, J., "Introductory Programming, Criterion-Referencing, and Bloom." *Proceedings of the SIGCSE 2002 Symposium.* Covington, KY. February-March 2002. pp.143-147.

[110] Lister, R., Fitzgerald, S., et al, "A multi-national study of Reading and Tracing Skills in Novice Programmers." *Proceedings of ITiCSE'04.* UK 2004. pp.119-150.

[111] López-Gaona, A., "The Importance of Design in CS1." *Inroads, The SIGCSE Bulletin*, June 2000. pp.53-55.

[112] Mayer, R. E., "The Psychology of Learning Basic." *Communications of the ACM.* November 1979.

[113] Mayer, R. E., and Bayman, P. "Psychology of Calculator Languages: a Framework for Describing Differences in Users' Knowledge." *Communications of the ACM.* August 1981.

[114] Mayer, R. E. et al., "Learning to Program and Learning to Think: What's the Connection." *Communications of the ACM.* July 1986.

[115] Mayer, R. E. (ed.) *Teaching and learning Computer Programing*, Lawrence Erlbaum Associates, Publishers. Hilldsdale, NJ. 1988.

[116] Mayer, R. E., *Thinking, Problem Solving, Cognition 2nd ed.*, W H Freeman & Co. 1992.

[117] McCracken, M., et al., "A Multi-national, Multi-institutional Study of Assessment of Programming Skills of First-year CS Students." *Proceedings of ITiCSE'01.* pp.125-140. 2001.

[118] McDonald, P., "The Nature of Algorithm Understanding: A Phenomenographic Investigation." *Doctoral Consortium. SIGCSE 2004 Symposium.* ACM. Norfolk, VA. March 2004.

[119] McIver, L., "The Effect of Programming Language on Error Rates of Novice Programmers." *12th Workshop of the Psychology of Programming Interest Group*. Conzenza, Italy. April, 2000. pp.181-192.

[120] McKinney, S. and Denton, L. F., "Houston, We Have a Problem: There's a Leak in the CS1 Affective Oxygen Tank." *Proceedings of the SIGCSE 2004 Symposium*. ACM. Norfolk, VA. March 2004. pp.236-239.

[121] Meyer, B., et al., "Empirical study of novice error paths." Unpublished technical report, August 2005. <http://se.ethz.ch/~meyer/publications/teaching/novices.pdf> Accessed on: December 19, 2006.

[122] National Research Council. *Being Fluent with Information Technology*. National Academy Press, 1999.

[123] NCS Pearsons. *Computer Programmer Aptitude Battery (CPAB)*. Reid London House, 1993.

[124] Palakal, M., et al., "An Interactive Learning Environment for Breadth-First Computing Science Curriculum." *Proceedings of the SIGCSE 1998 Symposium*. Atlanta, GA. February 1998.

[125] Pane, J.F., et al., "Studying the Language and Structure in Non-Programmer's Solutions to Programming Problems." *International Journal of Human-Computer Studies*. pp.237-264. Academic Press, 2001.

[126] Pane, J. F., "A Programming System for Children that is Designed for Usability." *PH.D. Thesis, CMU-CS-02-127.* Carnegie Mellon University. Pittsburgh, PA. May 3, 2002.

[127] Pea, R. D., "Language-Independent conceptual 'Bugs' in Novice Programming." *Journal of Educational Computing Research.* Vol. 2 No. 1, 1986.

[128] Perkins, D., et al., "Conditions of Learning in Novice Programmers." In: Soloway & Spohrer, *Studying the Novice Programmer,* 1989. pp.261-279.

[129] Petre, M., "Why Looking isn't Always Seeing: Readership Skills and Graphical Programming." *Communications of the ACM.* June 1995.

[130] Petre, M., Personal Communication. *Doctoral Consortium. SIGCSE 2004 Symposium.* ACM. Norfolk, VA. March 2004.

[131] Pillay, N., "Developing Intelligent Programming Tutors for Novice Programmers." *Inroads, The SIGCSE Bulletin,* June 2003. pp.78-82.

[132] Polya, G., *How to Solve it.* Princeton University Press. Princeton, NJ. 1945.

[133] Postner, L., "The Challenges of Learning to Program." *SIGCSE 2003 Doctoral Consortium.* 2003.

[134] Pressman, R. S., *Software Engineering: A Practitioner's Approach.* McGraw Hill, 2004

[135] Proulx, V. K., "Programming Patterns and Design Patterns in the Introductory Computer Science Course." *Proceedings of the SIGCSE 2000 Symposium.* Austin, TX. March 2000. pp.80-84.

[136] Rajlich, V. and Wilde, N., "The Role of Concepts In Program Comprehension." *Proceedings of IWPC 2002.* pp.271-278.

[137] Ramalingam, V. and Wiedenbeck, S., "An Empirical Study of Novice Program Comprehesion in the Imperative and Object-oriented Styles." *Seventh Workshop on Empirical Studies of Programmers*. Alexandria, Virginia. 1997. pp.124-139.

[138] Raymond, D. R., "Reading Source Code." *Proceedings of the 1991 CAS Conference*. IBM Canada Centre for Advanced Studies, Toronto, Ontario. October, 1991. pp.3-16.

[139] Rieman, J., "A Field Study of Exploratory Learning Strategies." *ACM Transactions on Computer-Human Interaction*. September 1996. pp.189-218.

[140] Rosso, A. and Daniele, M., "Our Method to Teach Algorithmic Development." *Inroads, The SIGCSE Bulletin*, June 2000. pp.49-52.

[141] Schou, C. D., et al., "Literary Criticism and Programming Pedagogy." *ACM publications*, 1988. pp. 67-71.

[142] Schwartz, J., "Forrester: Skills Shortage Will Worsen Unless Industry Seeds IT Talent." Dr. Dobb's Portal, Jun 13, 2006. <http://www.ddj.com/dept/global/189400869 > Last accessed: June 18, 2006.

[143] Selley, N., *The Art of Constructivist Teaching in Primary School*. David Fulton Publishers. London, UK. 1999.

[144] Seo, S. and Koro-Ljungberg, M., "A Hermeneutical Study of Older Korean Graduate Students' Experiences in American Higher Education: From Confucianism to Western Educational Values." *Journal of Studies in International Education*, Vol.9, No.2, Summer 2005. pp.164-187.

[145] Shaft, T. M., "Helping Programmers to Understand Computer Programs." *Data Base Advances*. November 1995. pp.25-46.

[146] Sheil, B. A., "A Psychological Study of Programming." *ACM Computing Surveys.* Vol. 13 No. 1, March 1981.

[147] Simon, S., et al., "The ability to articulate strategy as a predictor of programming skill," *Proceedings of the 8th Australian conference on Computing education.* Hobart, Australia, 2006. pp. 181-188.

[148] Skillprofiler Analytics. <http://www.skillprofiler.com/spdiff.asp> Accessed on December 27, 2006.

[149] Sleeman, D., "The Challenges of Teaching Computer Programming". *Communications of the ACM.* September 1986. pp. 840-841.

[150] Smith, P. and Webb, G. I., "Reinforcing a Generic Computer Model for Novice Programmers." *Proceedings of the Seventh Australian Society for Computers in Learning in Tertiary Education Conference (ASCILITE '95).* Melbourne, Australia. 1995.

[151] Snyder, L., "Computer Scientist Says all Students Should Learn to Think 'Algorithmically'." (Interview by F. Olsen.) *The Chronicle of Higher Education,* May 5, 2000. p.A49.

[152] Snyder, L., *Fluency with Information Technology,* Addison-Wesley, 2003.

[153] Soloway, E., "Learning to program = learning to construct mechanisms and explanations." *Communications of the ACM.* September 1986. pp.850-858.

[154] Soloway, E. and Spohrer, J. C. (eds.) *Studying the Novice Programmer,* Lawrence Erlbaum Associates, Publishers. Hilldsdale, NJ. 1989.

[155] Spohrer, J. C. and Soloway, E., "Novice Mistakes: Are the Folk Wisdoms Correct?" *Communications of the ACM.* July1986. pp.624-632.

[156] Sophatsathit, P., "A Study of Programming Skill Development in Education."

[157] Steffe, L. P. and Gale, J. (eds.) *Constructivism in Education.* Lawrence Erlbaum Associates, Publishers. Hilldsdale, NJ. 1995.

[158] Stone, J. A. and Madigan, E., "Inconsistencies and Disconnects." *Communications of the ACM.* April 2007. pp.76-79.

[159] Stringer, E., *Action Research in Education.* Prentice Hall. Upper Saddle River, NJ. 2004.

[160] Sydenham, P. H., *Systems Approach to Engineering Design.* Artech House. Boston, MA. 2004.

[161] Thomas, L., et al., "Learning Styles and Performance in the Introductory Programming Sequence." *Proceedings of the SIGCSE 2002 Symposium.* Covington, KY. February-March 2002. pp.33-37.

[162] Thompson, S., et al. *Problem Solving in General.* June 1996. <http://www.cs.kent.ac.uk/people/staff/djb/probSolving.html> Accessed on: January 27, 2004.

[163] Thompson, S., *How to Program it.* 1996. <http://www.cs.kent.ac.uk/people/staff/sjt/Haskell_craft/HowToProgIt.html> Accessed on: January 27, 2004.

[164] Thompson, E., et al., "Exploring Learner Conceptions of Programming." *Proceedings of the Australasian Computing Education Conference, ACE 2006.* Hobart, Australia. 2006.

[165] Tucker, A., et al., *Fundamentals of Computing I.* McGraw Hill. 1994.

[166] Vasconcelos, J., *Notas para el Curso Avanzado de Solución de Problemas y Programación.* Class Notes, 1998.

<http://www.cs.jhu.edu/~jorgev/cs106/ProblemSolving.html> Accessed on: December 12, 2006.

[167] Vasconcelos, J., *Manual de Contrucción de Programas.* University Digital Library, BDU. UNAM. México, 2000. <http://www.bibliodgsca.unam.mx/manuales/manual.pdf> Access on October 30, 2006.

[168] Vasconcelos, J. and Houlahan, J., "Development and Interpretation of a Survey on Problem Solving Skills." *Personal communication.* August 28, 2003.

[169] Vasconcelos, J., "A Course in Algorithmic Thinking." (Poster.) *SIGCSE 2004 Symposium.* Norfolk, VA. March 2004.

[170] von Glasersfeld, E., "A Constructivist Approach to Teaching." In *Constructivism in Education.* Steffe, L. P. and Gale, J. (eds.) Lawrence Erlbaum Associates, Publishers. Hilldsdale, NJ. 1995. pp. 3-15.

[171] von Glasersfeld, E., "Cognition, Construction of Knowledge, and Teaching." In *Constructivism in Science Education,* Matthews, M.R. (ed.) Kluwer Academic Publishers. Dordrech, Netherlands. 1998. pp.11-30.

[172] Wagner, T. A. and Harvey, R. J., "Developing a New Critical Thinking Test Using Item Response Theory". *SIOP 2003 Conference,* Society for Industrial and Organizational Psychology. Orlando, FL. 2003.

[173] Weimberg, G. M., *The Psychology of Computer Programming.* Van Nostrand Reinhold. NY, 1971.

[174] Weiner, L. H., "The Roots of Structured Programming." *ACM SIGCSE Bulletin,* June 1978. pp.243-254.

[175] Wen, J., et.al., *The Study of Information Aptitude Scale Development.* Institute of Information and Computer Education (internal document). Taiwan, 2000.

[176] Whalley, J.L., et al., "An Australasian study of reading and comprehension skills in novice programmers, using the bloom and SOLO taxonomies," *Proceedings of the 8th Australian conference on Computing education.* Hobart, Australia, 2006. pp. 243-252.

[177] Wiedenbeck, S., "Factors Affecting the Success of Non-Majors in Learning to Program." *Proceedings of ICER 2005.* Seattle, Washington. October 2005, pp.13-24.

[178] Winslow, L. E., "Programming Pedagogy—A psychological Overview." *ACM SIGCSE Bulletin,* September 1996. pp.17-25.

[179] Wirth, N., *Algorithms + Data Structures = Programs.* Prentice Hall. Englewood Cliffs, N.J. 1976. Wirth, Niklaus. "CS Education: The Road Not Taken." *ITiCSE 2001 Conference.* Aarhus, Denmark. ACM. June 2002. pp.1-3.

[180] Zahorian, S. A., et al., "Question Model for Intelligent Questioning Systems in Engineering Education." *Proceedings of the 31st ASEE/IEEE Frontiers in Education Conference.* October 2001. Reno, NV. 2001.

[181] Ziegler, U. and Crews, T., "An Integrated Program Development Tool for Teaching and Learning How to Program." *Proceedings of the SIGCSE 1999 Symposium.* New Orleans, LA. March 1999. pp.276-280.

Appendix A

Surveying Problem Solving

A.1. Introduction

To better appreciate preconceptions and preliminary skills new students had when starting a programming course, Vasconcelos and Houlahan designed a questionnaire comprising questions from five ability domains involved in solving problems algorithmically: *(i)* reading comprehension, *(ii)* problem identification, *(iii)* algebraic manipulation, *(iv)* stepwise planning, and *(v)* process analysis: tracing and debugging. Questions and problems were selected to resemble those usually found in introductory programming coursework but, rather than assessing knowledge or dexterity, they were intended to find error patterns and trends that could indicate fragile skills. The questionnaire developed is described in the following sections.

A.2. Reading Comprehension

This category was intended to test ability to read general texts and word problems objectively. The applicant was asked to answer several questions without involving

information not provided in the passage. Answers could indicate attention to details, inference abilities, preconceptions, knowledge of key terms, and proper use of quantifiers. (See figure A.1.)

Read this passage very carefully and mark as true (T) of false (F) the statements below. Select the answer that best matches the information given in the passage. Do not recur to any knowledge you may have of the topic.

Prime numbers fascinate and frustrate everyone who studies them. Their definition is so simple and obvious; it is so easy to find a new one; multiplicative decomposition is such a natural operation. Why, then, do primes resist attempts to order and regulate them strongly? Do they have no order at all or are we too blind to see it? There is, of course, some order hidden in the primes. The Sieve of Eratosthenes shakes the primes out of the integers. First 2 is a prime. Now knock out every higher even integer (which must all be divisible by 2). The next higher surviving integer, 3, must also be prime. Knock out all its multiples, and 5 survives. Knock out the multiples of 5, and 7 remains Keep on this way and each integer that falls through the sieve is a prime. This orderly if slow procedure will find every prime. Furthermore, as n goes to infinity, we know that ratio of primes to non-primes among the first n integers approaches ($\log_e n$) */n. Unfortunately, the limit is only statistical and does not actually help in finding primes.*

[Wheterell's Etudes]

T ( ) F ( ) All primes have a hidden order.
F. The statement uses universal quantifier.

T ( ) F ( ) The sieve takes advantage of Euclid's technique.
F. Information not provided.

T ( ) F ( ) The input required by the sieve is numbers.
T. Comprehension check.

T ( ) F ( ) By using the sieve, the resulting output are odd numbers.
T. The statement can be inferred from the text.

T ( ) F ( ) The passage describes the meaning of the prime numbers.
F. The statement has an incorrect interpretation.

T ( ) F ( ) The sieve detects any prime except 0.
F. Information not given in the text.

T ( ) F ( ) The input required by the sieve may be negative numbers.
F. Information not given in the text

T ( ) F ( ) In a sieve of m x n numbers, at least n^2 are prime.
F. Wrong deduction

T ( ) F ( ) By using the sieve, the resulting output values are primes
T. Directly from the text..

T ( ) F ( ) The sieve takes advantage of Eratosthenes' technique.
T. Directly from the text.

T ( ) F ( ) The word "blind", 4th line, means "lack of sight".
F. Unwanted meaning.

T ( ) F ( ) The word "blind", 4th line, means "clueless".
T. Deduced from text style

T ( ) F ( ) The passage describes **the** technique to find all the prime numbers.
F. Wrong quantifier (there is more than one technique).

T ( ) F ( ) The passage describes **one** technique to find all the prime numbers.
T. Correct quantifier.

FIGURE A.1. A Reading Comprehension Question.

Then, aiming to appreciate understanding of the reading, a subsequent question asked the applicant to summarize the passage using his/her own vocabulary. The answers could indicate writing skills, misunderstandings, and even reluctance to write.

A.3. Problem identification

Within this category there were questions to test applicant's ability to summarize and organize information, and to identify issues given a context. In addition, it also checks generalization of rules, usually expressed in the form of algebraic relations. Answers could indicate attention to word details, issues finding information, understanding the problem, or translating it to a different domain. (See figure A.2.)

A can of paint has a label that says one-gallon covers x square feet. You have to paint a cement block wall on its front side (only) with 2 coats of paint (you need to paint it twice). The wall is l feet long and h feet high and t feet think. What would you say to the person who asks you how to "figure out" the correct number of gallons to buy?

( ) **a)** Multiply l by h by t and divide by 2.
( ) **b)** Multiply l by h, h by t, add those numbers together and divide the last answer by x.
( ) **c)** Multiply 2 by l, that answer by x, and then divide by 2.
( ) **d)** Multiply 2 by h, that answer by l and then divide by x.

FIGURE A.2. A Problem Identification Question.

A.4. Algebraic Manipulation

The questions within this category tested applicant's ability to perform simple arithmetic operations, either numerically (fig. A.3) or in algebraic representation (fig. A.4), to foreseen algorithm output, and to interpret word problems.

Applicants' answer could indicate attention to numeric details, wording or formula confusion, and concrete or abstract thinking.

Your younger brother is planning a sleepover with 5 friends. Your mother told him to buy 2 hot dogs, 3 candy bars and something to read, for himself and each guest. He also needs some soda, and knows that 1 liter of soda is enough for 3 kids. How much food will he buy at the store?

( ) a) 2 Hotdogs, 3 candies, 1 soda, 5 Comics
( ) b) 10 Hotdogs, 15 candies, 2 sodas, 5 Comics
( ) c) 10 Hotdogs, 15 candies, 1 soda, 6 Comics
( ) d) 12 Hotdogs, 18 candies, 2 sodas, 6 Comics
( ) e) 12 Hotdogs, 18 candies, 2 sodas, 5 Comics

FIGURE A.3. A Problem on Arithmetic Skills with Numerical Answer.

What is the result of following these instructions?

Step 1: Think of a number, but keep it silently in your mind.
Step 2: Take your number and multiply it by 2
Step 3: Add 8 to the previous result.
Step 4: Take the result in step 3 and subtract the number you started with.
Step 5: What is the answer you got?

What is your answer? _____

FIGURE A.4. A Problem on Arithmetic Skills with Algebraic Answer.

A.5. Planning Strategy

This category seeks to elucidate applicant's ability to describe simple tasks in algorithmic fashion (fig. A.5). Answers can reveal levels of detail or abstraction, issues in thought expression, misconceptions on algorithms, causal logic skills, and proper use of assumptions. [21]

[21] This question was removed from the final survey because, after piloted, it showed to be of a higher level with respect to the rest of the survey.

Devise a strategy to calculate and report the average, maximum and minimum fuel efficiencies for a car, expressed as miles per gallon, given any given list of miles and gallons recorded for fill-ups, for example {(12,1), (140,7), (350,12), (240,12), ..., (*n* miles, *m* gallons)}. Write your detailed strategy in pseudocode form, if possible.

FIGURE A.5. A Problem on Planning.

A.6. Process Analysis

This section tested applicant's ability to work with short sequences of simple instructions (fig. A.6): understand purpose of instructions, hand-tracing, and debugging). The question is aimed to elucidate consistency in application of logic, misconceptions in algorithms, inadequate assumptions, and intuition in loops and conditionals.

You are given a program to display decimal values of 1/1, ½, 1/3, ¼, 1/5, however it does not work correctly. Find the problem(s) and make the necessary corrections.

```
Set num to 0
While num < 5
    Compute dec = 1/ num
    Display dec
   Add 1 to num
End While
```

FIGURE A.6. A Problem on Debugging.

A.7. Background

This section served to record applicant's preconceptions and understanding of computing fundamentals. It included a basic assessment of student's intuition and attitude towards programming and preferred learning "techniques" (fig. A.7.)

1. What does a computer know?
2. What is the meaning of A = A + 1?
3. When Jean cannot figure out the answer to a question on a multiple-choice test, she just chooses answer "b" because she has been told that this is a good method to use. Jean is relying upon _____ to answer the questions that she cannot figure out.
   (a) a guaranteed method
   (b) an algorithm
   (c) cognitive restructuring
   (d) a heuristic
   (e) inductive reasoning

FIGURE A.7. Several Questions on general background

A.8. Model Survey

Johns Hopkins University Department of Computer Science
Survey in Algorithmic Problem Solving

Name _____

List ALL Current Computer Science Courses, including sections:

_____

_____

Time Spent on this survey _____ (not including the last 3 questions)

The following questionnaire is intended to be a diagnostic tool to improve our classes based on your current abilities to solve problems. The results will not affect your final grade in any way. Please, do it on your own and be completely honest, do not use any book, notes or calculator, and do not correct any answer you have already written. If you do not understand the question, or do not know the answer, indicate so.

1. Read this passage very carefully and answer the following five questions.

Prime numbers fascinate and frustrate everyone who studies them. Their definition is so simple and obvious; it is so easy to find a new one; multiplicative decomposition is such a natural operation. Why, then, do primes resist attempts to order and regulate them strongly? Do they have no order at all or are we too blind to see it? There is, of course, some order hidden in the primes. The Sieve of Eratosthenes shakes the primes out of the integers. First 2 is a prime. Now knock out every higher even integer (which must all be divisible by 2). The next higher surviving integer, 3, must also be prime. Knock out all its multiples, and 5 survives. Knock out the multiples of 5, and 7 remains. Keep on this way and each integer that falls through the sieve is a prime. This orderly if slow procedure will find every prime. Furthermore, as n goes to infinity, we know that the ratio of primes to nonprimes among the first n integers approaches ($\log_e$ n) /n. Unfortunately, the limit is only statistical and does not actually help in finding primes. [Wheterell's Etudes]

For each of the following statements mark them as true (T) or false (F) according to the information given, or inferred from the text. Remember; select the answer that best matches the information previously given. (Do not base your answers on any prior knowledge.)

( ) a) All primes have a hidden order.
( ) b) The sieve takes advantage of Euclids' technique.
( ) c) The input required by the sieve is numbers.
( ) d) By using the sieve, the resulting output is odd numbers.
( ) e) The passage describes the meaning of the prime numbers.
( ) f) The sieve detects any prime except 0.

( ) g) The input required by the sieve may be negative numbers.
( ) h) In a sieve of m x n numbers, at least n^2 are prime.
( ) i) By using the sieve, the resulting output values are primes.
( ) j) The sieve takes advantage of Eratosthenes' technique.
( ) k) The word "blind", 4th line, means "lack of sight".
( ) l) The word "blind", 4th line, means "clueless".
( )m) The passage describes the technique to find all the prime numbers.
( ) n) The passage describes one technique to find all the prime numbers.

2. Rewrite the passage of question 1 using your own words. (Continue on back if necessary.)

3. What would this sequence of instructions accomplish?

Step 1: Multiply the price by .07.
Step 2: Add that answer to the price.

a) Calculates a 7% sales tax.
b) Calculates a 7% price reduction.
c) Calculates a total price including a 7% sales tax.
d) Calculates a markup that raises the price 107%.

4. A can of paint has a label that says one-gallon covers x square feet. You have to paint a cement block wall on its front side (only) with 2 coats of paint (you need to paint it twice). The wall is l feet long and h feet high and t feet think. What would you say to the person who asks you how to "figure out" the correct number of gallons to buy?

a) Multiply l by h by t and divide by 2.
b) Multiply l by h, h by t, add those numbers together and divide the last answer by x.
c) Multiply 2 by l, that answer by x, and then divide by 2.
d) Multiply 2 by h, that answer by l and then divide by x.
e)

5. How would you tell a younger person to find the total cost for gasoline for a trip of X miles with a car that gets Y miles per gallon, if gas costs Z dollars per gallon?

a) Divide X by Y, then divide the result by Z.
b) Multiply X by Y, then divide the result by Z.
c) Multiply X by Y, and then multiply the result by Z.
d) Divide X by Y, then multiply the result by Z.

6. What would this sequence of instructions accomplish?

Step 1: Divide 100 by 24.
Step 2: Round that answer up to the next larger whole number.

a) Calculates how many gallons of gas are used to go 100 miles.

b) Calculates how many vehicles are needed to transport 100 people if every vehicle carries 24 people.
c) Calculates how many boxes will be completely filled with apples if 100 apples are to be put in 24 boxes.
d) All of the above.

7. Devise a strategy to calculate and report the average, maximum and minimum fuel efficiencies for a car, expressed as miles per gallon, given any given list of miles and gallons recorded for fill-ups, for example {(12,1),(140,7),(350,12),(240,12),..., (n miles, m gallons)}. Write your detailed strategy in pseudocode form, if possible.

8. Your younger brother is planning a sleepover with 5 friends. Your mother told him to buy 2 hot dogs, 3 candy bars and something to read, for himself and each guest. He also needs some soda, and knows that 1 liter of soda is enough for 3 kids. How much food will he buy at the store?

a) 2 Hotdogs, 3 candies, 1 soda, 5 Comics
b) 10 Hotdogs, 15 candies, 2 sodas, 5 Comics
c) 10 Hotdogs, 15 candies, 1 soda, 6 Comics
d) 12 Hotdogs, 18 candies, 2 sodas, 6 Comics
e) 12 Hotdogs, 18 candies, 2 sodas, 5 Comics

9. What is the result of following these instructions?
Step 1: Think of a number, but keep it silently in your mind.
Step 2: Take your number and multiply it by 2
Step 3: Add 8 to the previous result.
Step 4: Take the result in step 3 and subtract the number you started with.
Step 5: Write down your answer _____

10. You are given a program to display decimal values of 1/1, ½, 1/3, ¼, 1/5, however it does not work correctly. Find the problem(s) and make the necessary corrections.

Set num to 0

While num < 5

Compute dec = 1/ num

Display dec

Add 1 to num

End While

11. These instructions should display the even numbers between 1 and 10 inclusive, in descending order. Verify for correctness. If this is the case, explain the procedure, otherwise, explain the problem(s).

Set num to 10
Repeat
  Subtract 2 from num
Until num = 1

12. Please answer the following questions to the best of your abilities.

12.1 What does a computer know?

_____

_____

_____

12.2 What is the meaning of A = A + 1?

_____

_____

_____

12.3 Describe in detail the action that the simplest Print command would have.

_____

_____

_____

12.4 Describe in detail the action that the simplest Read command would have.

_____

_____

_____

12.5 When Jean cannot figure out the answer to a question on a multiple choice test, she just chooses answer "b" because she has been told that this is a good method to use. Jean is relying upon _____ to answer the questions that she cannot figure out.

a) a guaranteed method
b) an algorithm
c) cognitive restructuring
d) a heuristic
e) inductive reasoning

12.6 Dave is having trouble coming up with the answer to a math problem. He decides to look through the book and finds the step-by-step procedure for solving the problem. If he follows the steps in the book, he is using: _____

e) subgoals
f) working backwards
g) an algorithm
h) a heuristic
i) chunking

12.7 I prefer to (Select only one option per row)

a)	( ) Solve a puzzle	( ) Skip the puzzle
b)	( ) Build a model	( ) Buy a built model
c)	( ) Understand laws of Physics	( ) Know formulas used in Physics
d)	( ) Read every paragraph in a paper	( ) Read abstract and conclusion
e)	( ) Follow procedures when dealing with new situations	( ) Improvise in case of new problems

12.8 Why do you think it is important to learn computer programming?

_____

_____

_____

12.9 Why do you think it is important to learn algorithm design?

_____

_____

_____

12.10 What is the best way an instructor can get your attention? (Select all that apply)

a) By presenting funny facts.
b) By being argumentative.

c) By being just informative.
d) By drawing pictures.
e) By asking questions.

(The following questions are under testing for use in future surveys. Do NOT include the time it takes to answer them in your reported survey time.)

a. Between the points A and B there is a distance of 5 miles, and you are able to run extremely fast, so you can go from A to B instantly, what is your speed. (Remember, speed = distance / time)

_____

_____

_____

b. A gas pump has being tested. Results show that every time the machine dispenses fuel, it delivers two gallons less than displayed. What is the problem?

_____

_____

_____

c. A rocket is scheduled for launching when a timer reaches the value of 0. If the control software has this line "IF timer < 0 THEN release_clamps", when does the lift off occur? Why?

_____

_____

A.9. Analysis of Survey Administered

Johns Hopkins University Department of Computer Science
Survey in Algorithmic Problem Solving

Name **J. H.**

List ALL Current Computer Science Courses, including sections:

_____

_____

Time Spent on this survey _____ (not including the last 3 questions)

The following questionnaire is intended to be a diagnostic tool to improve our classes based on your current abilities to solve problems. The results will not affect your final grade in any way. Please, do it on your own and be completely honest, do not use any book, notes or calculator, and do not correct any answer you have already written. If you do not understand the question, or do not know the answer, indicate so.

Purpose: Test "reading comprehension" in CS-related texts or word- problems without the use of information not provided in the passage.

{The survey designer has to select a clear, short passage (from an already published source) containing identifiable quantifiers, keywords, possibility to infer information or to be connected with applicant current knowledge, possibility of being misinterpreted because of poor reading, and context-dependent words ..., that the applicant is supposed to detect in order answer the questions correctly.}

1. Read this passage very carefully and answer the following five questions.

Prime numbers fascinate and frustrate everyone who studies them. Their definition is so simple and obvious; it is so easy to find a new one; multiplicative decomposition is such a natural operation. Why, then, do primes resist attempts to order and regulate them strongly? Do they have no order at all or are we too blind to see it? There is, of course, some order hidden in the primes. The Sieve of Eratosthenes shakes the primes out of the integers. First 2 is a prime. Now knock out every higher even integer (which must all be divisible by 2). The next higher surviving integer, 3, must also be prime. Knock out all its multiples, and 5 survives. Knock out the multiples of 5, and 7 remains. Keep on this way and each integer that falls through the sieve is a prime. This orderly if slow procedure will find every prime. Furthermore, as n goes to infinity, we know that the ratio of primes to nonprimes among the first n integers approaches ($\log_e$ n) /n. Unfortunately, the limit is only statistical and does not actually help in finding primes. [Wheterell's Etudes]

For each of the following statements mark them as true (T) or false (F) according to the information given, or inferred from the text. Remember; select the answer that

best matches the information previously given. (Do not base your answers on any prior knowledge.)

(T) a) All primes have a hidden order.
F. Quantifiers misinterpreted.
(T) b) The sieve takes advantage of Euclids' technique.
F. Information not provided.
(T) c) The input required by the sieve is numbers.
(F) d) By using the sieve, the resulting output is odd numbers.
T. Answer must be inferred from text.
(F) e) The passage describes the meaning of the prime numbers.
(T) f) The sieve detects any prime except 0.
F. Information not given in the text.
(F) g) The input required by the sieve may be negative numbers.
(F) h) In a sieve of m x n numbers, at least n^2 are prime.
(T) i) By using the sieve, the resulting output values are primes.
(T) j) The sieve takes advantage of Eratosthenes' technique.
(F) k) The word "blind", 4th line, means "lack of sight".
(T) l) The word "blind", 4th line, means "clueless".
(T)m) The passage describes the technique to find all the prime numbers.
F. Wrong quantifier (there is more than one technique).
(T) n) The passage describes one technique to find all the prime numbers.

Probable tendency to perform quantum reading or "fast reading".

2. Rewrite the passage of question 1 using your own words. (Continue on back if necessary.)

The Sieve of Eratosthenes is used to find all the prime numbers out of the integers.

Purpose: Verify applicant's reading comprehension (understanding) by rewriting the passage using his/her own vocabulary.

Poor rephrase may indicate lack of reading skills or understanding. (There is no description of the method, It's more like a title.)

3. What would this sequence of instructions accomplish?

Step 1: Multiply the price by .07.
Step 2: Add that answer to the price.

j) Calculates a 7% sales tax.
k) Calculates a 7% price reduction.
l) Calculates a total price including a 7% sales tax.
m) Calculates a markup that raises the price 107%.

Purpose: Interpreting a [sequential] list of basic actions (instructions).

Categories of issues on algorithms interpretation detected through this question:
a. *Question was answered right [OK], procedure purpose was identified, either by just reading the steps or by understanding each step and the sequence as a whole.*

4. A can of paint has a label that says one-gallon covers x square feet. You have to paint a cement block wall on its front side (only) with 2 coats of paint (you need to paint it twice). The wall is l feet long and h feet high and t feet think. What would you say to the person who asks you how to "figure out" the correct number of gallons to buy?

f) Multiply l by h by t and divide by 2.
g) Multiply l by h, h by t, add those numbers together and divide the last answer by x.
h) Multiply 2 by l, that answer by x, and then divide by 2.
i) Multiply 2 by h, that answer by l and then divide by x.

5. How would you tell a younger person to find the total cost for gasoline for a trip of X miles with a car that gets Y miles per gallon, if gas costs Z dollars per gallon?

e) Divide X by Y, then divide the result by Z.
f) Multiply X by Y, then divide the result by Z.
g) Multiply X by Y, and then multiply the result by Z.
h) Divide X by Y, then multiply the result by Z.

6. What would this sequence of instructions accomplish?

Step 1: Divide 100 by 24.
Step 2: Round that answer up to the next larger whole number.

e) Calculates how many gallons of gas are used to go 100 miles.
f) Calculates how many vehicles are needed to transport 100 people if every vehicle carries 24 people.
g) Calculates how many boxes will be completely filled with apples if 100 apples are to be put in 24 boxes.
h) All of the above.

7. Devise a strategy to calculate and report the average, maximum and minimum fuel efficiencies for a car, expressed as miles per gallon, given any given list of miles and gallons recorded for fill-ups, for example {(12,1),(140,7),(350,12),(240,12),..., (n miles, m gallons)}. Write your detailed strategy in pseudocode form, if possible.

Search the minimum miles value in the list
Search the maximum miles value in the list

Purpose: Ability to describe some simple task in algorithmic way, and to elucidate the depth of detail, order and abstraction in the answer.

General idea., but without strategy or details.

8. Your younger brother is planning a sleepover with 5 friends. Your mother told him to buy 2 hot dogs, 3 candy bars and something to read, for himself and each guest. He also needs some soda, and knows that 1 liter of soda is enough for 3 kids. How much food will he buy at the store?

f) 2 Hotdogs, 3 candies, 1 soda, 5 Comics

g) 10 Hotdogs, 15 candies, 2 sodas, 5 Comics
h) 10 Hotdogs, 15 candies, 1 soda, 6 Comics
i) 12 Hotdogs, 18 candies, 2 sodas, 6 Comics
j) 12 Hotdogs, 18 candies, 2 sodas, 5 Comics

9. What is the result of following these instructions?
Step 1: Think of a number, but keep it silently in your mind.
Step 2: Take your number and multiply it by 2
Step 3: Add 8 to the previous result.
Step 4: Take the result in step 3 and subtract the number you started with.
Step 5: Write down your answer _____
13

Purpose: Test ability to hand-trace and express answer in algebraic form.

Answer not in algebraic form: x+8

10. You are given a program to display decimal values of 1/1, ½, 1/3, ¼, 1/5, however it does not work correctly. Find the problem(s) and make the necessary corrections.

Set num to 1

While num <= 5

Compute dec = 1/ num

Display dec

Add 1 to num

End While

11. These instructions should display the even numbers between 1 and 10 inclusive, in descending order. Verify for correctness. If this is the case, explain the procedure, otherwise, explain the problem(s).

Set num to 10
Repeat
Subtract 2 from num
Until num = 1

Until condition is not reached (10,8,6,4,2,0) condition must be changed to num=0

Problem not explained.

12. Please answer the following questions to the best of your abilities.

12.1 What does a computer know?

__computer knows how to do something specified by a program written by a human_

*Anwsered correctly at a high level*_____

_____

12.2 What is the meaning of A = A + 1?

incerment the value of A BY one

_____

_____

12.3 Describe in detail the action that the simplest Print command would have.

prints a character string in the monitor screen

*Details?*_____

_____

12.4 Describe in detail the action that the simplest Read command would have.

gets a character out of the keyboard to assign it to a variable

*Details?*_____

_____

12.5 When Jean cannot figure out the answer to a question on a multiple choice test, she just chooses answer "b" because she has been told that this is a good method to use. Jean is relying upon _____ to answer the questions that she cannot figure out.

f) a guaranteed method
g) an algorithm
h) cognitive restructuring
i) a heuristic
j) inductive reasoning

12.6 Dave is having trouble coming up with the answer to a math problem. He decides to look through the book and finds the step-by-step procedure for solving the problem. If he follows the steps in the book, he is using: _____

n) subgoals
o) working backwards

p) an algorithm
q) a heuristic
r) chunking

12.7 I prefer to (Select only one option per row)

f)	(X) Solve a puzzle	( ) Skip the puzzle
g)	(X) Build a model	( ) Buy a built model
h)	(X) Understand laws of Physics	( ) Know formulas used in Physics
i)	(X) Read every paragraph in a paper	( ) Read abstract and conclusion
j)	(X) Follow procedures when dealing with new situations	( ) Improvise in case of new problems

12.8 Why do you think it is important to learn computer programming?

TO GET ADVANTAGE OF COMPUTER TECHNOLOGIES AND DEVELOP SOLUTIONS OF MY OWN

(Personal interest)

12.9 Why do you think it is important to learn algorithm design?

TO LEARN TO COMMUNICATE SOLUTIONS TO PROBLEMS
(Documentation, teaching?)

12.10 What is the best way an instructor can get your attention? (Select all that apply)

f) By presenting funny facts.
g) By being argumentative.
h) By being just informative.
i) By drawing pictures.
j) By asking questions.

(The following questions are under testing for use in future surveys. Do NOT include the time it takes to answer them in your reported survey time.)

a. Between the points A and B there is a distance of 5 miles, and you are able to run extremely fast, so you can go from A to B instantly, what is your speed. (Remember, speed = distance / time)

infinitum_(5/t, when t->0)

Is it possible to have an infinite speed?

b. A gas pump has being tested. Results show that every time the machine dispenses fuel, it delivers two gallons less than displayed. What is the problem?

the tube retains the fuel

dispatching system is altered

Simplest answer. Also, the gas counter may be off by two.

c. A rocket is scheduled for launching when a timer reaches the value of 0. If the control software has this line "IF timer < 0 THEN release_clamps", when does the lift off occur? Why?

1 second later than zero

Never, ignition system starts without the clamps ever being released.

Vita

Jorge Vasconcelos was born in México City in 1971. He graduated with a Bachelor of Science in Computer Engineering from the Arturo Rosenblueth Foundation in 1996. He entered the Department of Computer Science at the Johns Hopkins University, receiving a Master of Science and Engineering in 2000, and has been working in Computer Science Education towards a Ph.D. His research work has moved within four trends: (*i*) application of the constructivist paradigm to computer science education, (*ii*) creation of instruments for early detection of cognitive abilities required in algorithmic problem solving, (*iii*) study of individual differences aiming to better differentiate instruction within the computer science classroom, and (*iv*) design of curricular models focused in problem solving and algorithmic thinking. His current work was awarded honorable mention in the Sixth Graduate Research Symposium at The College of William and Mary. He has taught introductory computer science for 17 years and published several textbooks for Mexican high schools.