University of Nevada
Reno

The Five Step Programming Process

A thesis submitted in partial fulfillment of the
requirements for the degree of
Master of Science
in Computer Science

by

Michael Ernest Leverington

Frederick C. Harris, Jr., PhD./Thesis Advisor

December 2010

THE GRADUATE SCHOOL

We recommend that the thesis
prepared under our supervision by

MICHAEL ERNEST LEVERINGTON

entitled

The Five Step Programming Process

be accepted in partial fulfillment of the
requirements for the degree of

MASTER OF SCIENCE

Frederick C. Harris, Jr., Ph. D., Advisor

Yaakov Varol, Ph. D., Committee Member

Jennifer Mahon, Ph. D., Graduate School Representative

Marsha H. Read, Ph. D., Associate Dean, Graduate School

December, 2010

Abstract

While educators have been working to teach introductory programming for between 40 and 50 years, the research has not supported either the value or the effectiveness of this endeavor. Recent research has found that teaching some thinking components along with the programming actions can provide positive results. A pilot study was conducted with college students learning a formalized and structured programming process in order to evaluate the efficacy of the process. Some positive, but limited, results were found from the study, although one of the interesting results discovered directly from students' responses is the need to further support this kind of activity. Preliminary results are reported, and opportunities for further study are identified.

Acknowledgements

I must first acknowledge my wife Sheryl who has been paying most of the bills for the past six years, and the support of my family of furry critters who help me keep perspective every day.

I also wish to thank Yaakov Varol for his exceptional support of my endeavors as a Computer Science student, researcher, and educator; and I must particularly thank my Advisors Jenny Mahon and Fred Harris for their constant and ongoing support of what has become a high quality education.

Contents

List of Figures

Chapter 1
Introduction

There are many opinions posited about the quality and effectiveness of teaching introductory computer programming, commonly called Computer Science I or CS1. However, one of the boldest and as it turns out, one of the best supported with research is that "many students do not know how to program at the conclusion of their introductory courses" [39]. This is a pretty strong indictment considering that computer programming is the first or second course taught in almost all Computer Science programs around the country. Another strong statement made by Robins, Rountree, and Rountree who recently reviewed the literature on learning and teaching programming is that "the average student does not make much progress in an introductory programming course" [52]. This same review quoted several others who offered comparable commentary (see for example, list in [52], p156).

The news is not all bad in this arena because there has been a significant amount of research conducted in the past three decades. The beginnings of this research were based on the idea that learning to program would help with students acquiring cognitive skills "such as planning abilities, problem-solving heuristics, and reflectiveness" [44], or "proper habits of mind" [38]. These results seem intuitively obvious, although Pea and Kurland went on to compare the efficacy of this kind of learning to that of Latin or other disciplines such as mathematics or logic that were previously thought to improve

students' minds. The other problem, and the one addressed in this thesis is that, as Pears, *et al.* [45] argued, the research -- active as it may be -- is not making inroads into classroom teaching practices.

The response to the present state of this discipline is to seek ways to make introductory computer programming learning more effective by attempting to move the research into the classroom. For example, cognitive researchers who have been studying memory organization in the form of *chunks* have provided a vehicle for researchers who have studied the "content and structure of programming knowledge" [52]. The memory chunk is a construct that explains how humans can hold individual items of information (e.g., names, ages, telephone numbers, change amounts, etc.), or large-scale semantically organized procedures (e.g., how to sum or average a list of numbers, how to find a maximum value in a list, etc.), sometimes called plans or *schemas* [41]. The concept of chunking, and to a significant extent the limitations of human memory chunks, is an important driver to the research reported herein.

Beyond the fundamentals of memory content or management, there is also the issue of what researchers should be studying in order to support improved programming learning. Davies [22] noted that just having content knowledge of programming may not be enough; he argued that a strategic model of knowledge representation is important, not just for organizing present knowledge, but for understanding the changes that occur as expertise develops. Thus, beyond teaching with consideration for human memory limitations, this research will also attempt to address the systematic organization of learner program development as both the learners and the programs evolve.

The third leg of the research driving the present study is related to the original argument, or question, related to the integration of teaching thinking processes with programming. Kirkwood [36] worked to integrate higher-order thinking and problem solving with a secondary school programming course. The results of her research were that the students found success at the programming endeavor by demonstrating well-designed programs, experiencing few syntactical or logical errors, and so on. In addition however, students also found success at the higher-level thinking endeavors with specifically improved cognitive and metacognitive characteristics.

Thuné and Eckerdal [61] conducted a qualitative analysis of student perceptions and found, among others, that "Computer programming is seen as a way of thinking, to solve problems, leading to the production of computer programs such as those that appear in everyday life. In addition, computer programming is experienced as a skill that can be used outside the programming course, and for other purposes than computer programming" [61]. Another study observing student learning included problem solving as one of the ways students go about learning to program [14], and still another study that included a lighter pre-CS1 course still included the learning of problem-solving abilities [2].

To summarize, even with a significant amount of research being conducted at present, the CS1 classroom does not seem to be benefiting from these endeavors. This thesis will report on a pilot study using a process that is driven by research in learner memory management, using the chunking construct, and other research-driven components. The process also seeks to provide a systematic vehicle that should support both better quality programming and improved student success with the development of

programs. And finally, the process attempts to teach and to apply higher-level thinking abilities in students so that their problem-solving abilities are improved, both within the CS1 classroom and without.

This thesis continues in Chapter 2 with extended background research on the issues and problems found in teaching and learning computer programming in CS1 courses, and then a review in Chapter 3 that includes the apparent needs identified to move the introductory programming teaching and learning discipline forward. In Chapter 4, the Five Step Programming Process will be proposed and in Chapter 5, a pilot study using this process will be evaluated for student usability and efficacy. Finally in Chapter 6, research conclusions from the pilot study will be presented, and future work that will include using this process in larger-scale studies will be proposed.

Chapter 2
Issues with CS1 Teaching

This chapter begins the discussion with some history and background of introductory computer programming education, and some of the more significant issues that have been researched related to CS1 teaching and learning.

2.1 Attitudes About Programming

While the number of Computer Science majors has experienced a small uptick in the past couple of years, it had declined by about 50% between 2002 and 2007 [55]. Failure rates vary but have been reported at 30% during that time [60]. This is unfortunate for two reasons: 1) computer programming is only part of the Computer Science discipline, but since students have to program to get through most CS courses, they must get started early; and 2) these kinds of losses make the future bleak for both industrial and academic institutions that need the kind of talent and fresh ideas that CS students can bring to the discipline.

Research from Simon, *et al.* [55] found that about 52% of about 300 CS majors made positive statements about taking CS1, but 26% provided negative responses. By itself, that statistic does not seem to bode well. However, when non-majors reported their opinions, only 45% provided positive responses and more than a third (36%) provided

negative responses. Within the total group, 22.9% of the responses stated that the course was hard or difficult, 15.3% made other generally negative statements, and 10.3% of the respondents reporting that the process was frustrating or stressful. While there were several other response categories, the last one of interest reported here is that 8.6% of the students reported that the course required a lot of time.

Other research can be used to show student frustration or student departure from Computer Science as a result of the introductory programming courses. Beaubouef and Mason [7] discussed the loss of students and summarize a variety of problems such as poor advising, poor math and problem-solving skills on the students' parts, poor lab courses and teaching issues on the institutions' parts, and so on. However, Computer Science and computer programming specifically can be really enjoyable when students experience successful interactions with their work. The reality found in the research is that this does not appear to be happening in the CS1 courses.

2.2 Cognitive Issues in Programming

Assuming that most students who take the CS1 course approach it with a belief that they will be successful, what kinds of problems could either frustrate them and/or stop them from continuing or striving in a course where many others have previously succeeded? One of the first considerations is cognitive ability. This is not meant to say that differential cognitive abilities are what make or break an introductory CS student's success in the course. Instead, it is analyzing the teaching process from the learners' cognitive characteristics.

Cognitive load is one of the key terms addressed in this study. This is the amount of processing ability needed by an individual at a given moment under given circumstances. The concept of cognitive load grew out of the need to model the access and use of limited real-time memory storage known as *working memory* [6]. While there is significant argument related to the size of working memory, the most recent research on this places the size at about four chunks (see [21] and associated articles for a comprehensive discussion of this topic). This would be a significant restriction that should be considered by any educator of any course, but when the educator is teaching a process that solves problems in a sequential and formal way using a language previously unknown to students in the programming environment that includes managing source code, compiled programs, and related data, this limitation is more than significant; it is critical.

Unfortunately, it is generally agreed that the memory capacity decreases when interactive elements such as, for example: using variables, in order to → conduct math, in order to → display a result, are combined. Paas, Renkl, and Sweller [42] and Sweller, van Merrienboer, and Paas [59] argued that working memory can only hold two or three novel interacting elements at a given moment. Adding to this issue, Yuen [65] argued that cognitive load is additive, which makes the addition of learning actions to the core learning quantity part of the problem. As early as 1985, Anderson and Jeffries studying students using the LISP programming language concluded that mistakes were made as a result of loss of working memory information [4]. This means that teaching the programming process must be highly refined to respond to this issue, and it also provides evidence as to why it may not have been done as well up to this point.

The good news about the limit is that once a human does integrate or internalize a concept or quantity of knowledge, it is moved toward the long-term memory area, and can then be recalled on command. This includes more complicated structures than simply memorized values which can then be recalled, and stored, as a chunk in working memory [52]. This can lead to the goal of this study by synthesizing a path that includes "a starting cue, a direction, a level, and a type of link to explore next" [52]. If this direct implementation of memory access can be used, students can learn to develop a program in concert with their memory operations as opposed to in conflict with them. Rist [51] used this approach when discussing how novices can transition to expert-like characteristics with both consideration for memory retrieval and program development.

Green, Bellamy, and Parker [32] created what they called the "parsing-gnisrap" model which, in the gnisrap condition, builds the program from known schemas found in memory; and in the parsing condition, works the memory process by essentially creating a large chunk representing a whole task or programming, and then taking the large chunk down to its pieces to understand the program. Ormerod [41] continued this discussion by referring to propositional representations formed in memory that can again hold whole plans, called scripts or schemas, and Rist discussed the evolution of what he calls a "plan schema" [51]. This is not to say that whole programs can be retrieved and replayed, but blocks or modules can be retrieved as components and programs can be created "opportunistically and incrementally" [31]. While this proposal takes advantage of human thought, it is clearly more of an expert action considering the assumption that a programmer has these blocks of schemas available to working memory. This is where educators must seek to make better novices rather than trying to create experts

considering that novices are working from a nearly empty set of retrievable modules. The expert-novice issue will be discussed later in this chapter.

Nevertheless, supporting the learning by novices leads to the issue discussed by Merrill [40] that teaching anything of complexity cannot be left to the learner, or as he puts it, "receiving little or no guidance (sink-or-swim) is not effective" [40]. As Merrill argued, students must be carefully guided from the simple to the complex. As one example of this process in the course of moving the student toward increasing skill, it should be noted that the translation of information into schemas (i.e., working chunks) takes more than an individual experience. The process of converting learned schemas into storable chunks requires a conscious effort and must be learned to a point where its parts do not have to be separately organized and manipulated. Sweller points out that this *automaticity* is required in order to bypass the limits of working memory and may require "extensive practice" [59]. While this is a requirement of managing the cognitive load/memory limitation issue, it is not difficult to overcome primarily with practice dedicated to the particular activity that is to be encoded as a schema. However, it is incumbent on the educator to see to the successful implementation of this activity.

Complexity and quantity of information can overwhelm a student who in the course of trying to manage and manipulate information in the limited working memory resources experiences *cognitive overload* [63]. This can be resolved by teaching whole programs that are light on details, and then breaking these holistic components down to comparable modules that as individual components are not cognitively expensive, but help the student gain the important schemas required for larger-scale components one at a

time. This is a critical component of the process proposed in this thesis. van Merrienboer, Kirschner, and Kester concluded their article with the following:

"limited working memory is no doubt the most central aspect of human cognitive architecture. There are many factors that an instructional designer must consider, but the cognitive load imposed by instructional designs should be the preeminent consideration when determining design structures" [63].

As part of developing the background for a programming/learning process, it should also be noted that the addition of this kind of process, much like the learning of so-called weak problem-solving skills such as top-down, means-ends, and so on, will also cost working memory or cognitive load overhead [63], a condition reiterated by Sweller, van Merrienboer, and Paas [59]. This is an issue that must be considered during the development of this kind of process, and it will need to be resolved in order to make the process effective.

All of the above issues come down to the management or limitation of cognitive load and the potential manipulation of the memory chunks holding those pieces of knowledge that will support the solution to a given problem. Gerjets, Scheiter, and Catrambone [25] argued for a modular approach to the learning, as opposed to what they called a "molar" view that attempts to identify and teach toward problem categories or patterns, which of course cost more cognitive load. This is an important consideration for an educator who wishes to place a learning tool in the hands of novice students; it is also a very convenient one for a programming teacher. Modularity is one of the most powerful and appropriate ways to support reliability and fault tolerance [37] in a program; and

using modularity has the potential to align student learning effectiveness with proper program development in a natural way.

For the cognitive issues identified in this section, there seem to be more problems than there are solutions. However, it is the identification of these issues that is important. Once these are known and/or made explicit, they can be addressed. The limitations of memory and cognitive load are severe, but knowing about them and adapting the learning to these limitations is really only a matter of scaling. It is admittedly a small scale, but it is nevertheless a somewhat quantifiable entity, and it should be possible to work with this.

2.3 Perception Issues in Programming

While it is not as significant or possibly well-recognized as the cognitive issue, human perception specifically related to programming is still a consideration. David Gilmore's [28] early work identified program organization characteristics that he showed to have an impact on program understanding by students. As early as 1980, Hartley [33] found that the use of white space (i.e., areas of programming text without any characters or text) has the potential to improve reader comprehension. This is important to students trying to manipulate interrelated items in their working memory, and could lead to assistance with organizing the information in the structured way required for moving program actions into long term memory as schemas.

Payne, Sime, and Green [43] showed that even a simple change in the perceptual characteristic of program text such as capitalizing some of the keywords significantly reduced errors. Gilmore [28] went on to provide signs that the comprehension of text and

other materials can be improved by clearly showing the underlying structure of the information. Gilmore's own research found that "structural visibility enables a general improvement in performance through a reduction in the demands on cognitive processes" [28].

One of the implications of Gilmore's research was that while languages themselves may not contribute to reduced cognitive load, the appropriate structural use of the language can drive increased programmer performance. In his conclusion, Gilmore argued as a result of his research that this improved performance can be a result of improved perceptual processing, improved programmer understanding, and an understanding of how to use a language's organization to drive these improvements.

To reiterate, the organization and presentation of text in a source code file is not as big a consideration as cognitive load and memory limitations. Nonetheless, it does exist, it has been researched to some extent, and it is a way to make inroads on program understanding and processing or cognitive load reduction. This part of the research will contribute to the proposed solution.

2.4 Focal Quantity Issues in Programming

The last component studied under the umbrella of cognitive processes is, like text structure, not as widely researched as for example, cognitive load. However, the recognition of this cognitive quantity is still important. Rist [51] found that if students did not already have a schema available to them, they would identify a piece of the problem -- essentially a short-term goal -- as a focal point, and then expand outward from the focal

point generating solution components necessary for solving the short-term goal. His label for this process was *plan creation*.

Rist found that this was a natural result for novice programmers who did not have experience in the form of a schema to use in a top-down programming process. He also noted that once this schema was developed, it would be re-used by the novice when a comparable, but new or novel problem was presented later. Rist called this "the start of detailed design in the domain of the program" [51]. In a later paper, Davies pointed out that "expert programmers were seen to generate significantly more focal lines during the early stages of the development of a program, whereas novice programmers generated significantly more non-focal lines" [22]. The important thing to note here is that these were essentially observations of the natural course of program development, without any attempt to modify either the expert or the novice approach. Another key point in following these observations was Davies' note that the "focal lines may represent a discrete level of design abstraction" [22].

Robins, Rountree, and Rountree [52] summed this up by saying that the best evidence that novices do not have the needed schemas is that they struggle with the appropriate use of focal design. Once again however, this commentary and the research supporting it is generally observational as to the natural order of beginning and experienced programmers. Understanding the value of focal components of a program, and knowing that they can be identified and exposed to student programmers provides another consideration for how to teach the process.

2.5 Expert/Novice Issues in Programming

A chapter on introductory programming issues would not be complete without consideration for the large amount of research in the area of experts and novices. To this point in the chapter, attitudinal and cognitive issues have been considered, as have issues related to perception and to conditions that drive or focus programmers' next steps. With these fundamental components provided, the broader -- but still primarily observational -- delineation between people with a significant number of discipline-specific skills, called experts, and those first learning a discipline, called novices, must be provided. This discussion will be presented here.

The novice approach to programming is considered to be superficial, tending toward a line-by-line development strategy. They do not have a complete picture -- called a mental model -- of the problem or solution in mind, and they tend to use superficial strategies or general problem-solving strategies that may or may not fit a specific problem [64]. For an individual with limited historical experiences and/or operational and retained schemas, this would seem appropriate. Novices tend to focus on the local or concrete components of a program [52], and while novices might be considering the syntax, experts tend to view the organization [28].

It would seem obvious then that as novices are focusing on the details, they will have trouble gaining an abstract view of their program [51]. With this limitation, they would obviously have a difficult time creating program plans -- as experts do -- and/or synthesizing a program solution [50, 52, 64]. Beyond simply causing problems with program development, the novices' inabilities to abstract segments of problem solution or

code means that they will have difficulty thinking their way through or test-running larger segments of their program or solution as experts do [51]. Rist's research went on to show that novices move through programs in a depth-first modality while experts move through them in a breadth-first way. Rist posited that this may be the novices' way to manage cognitive load [51].

Pennington [46] also conducted research on comprehension strategies, finding that individuals who were more able to comprehend programs provided more vague statements about the program and fewer details then individuals who were less competent at comprehending the programs. Her research tended to support Brooks' model that relies on an effective mapping between the problem and the program [13].

Brooks also posited that program comprehension is studied by experts as a top-down and hypothesis-driven activity. This concept -- as a model -- works for both experts and novices with the difference that experts will have an abstracted "view" of the larger program where novices will tend to have a narrow, or again, line-by-line view. When experts read programs, they chunk components into schema groups and can generate a more abstract or large-scale view of the program. When novices attempt this, they do not have the library of schemas as support, and as a result of that, cannot comprehend the whole program as the abstraction it could be.

As Robins, Rountree, and Rountree noted, "Experts can typically retrieve relevant plans from memory" and "Novices must typically create plans" [52]. This could make programs that may have the same general characteristics or patterns all look unique to the novice programmer. Beyond simply understanding a program, even when novices begin to understand parts of programming processing, Rogalski and Samurçay [53] posited that

novices continue to have difficulty with data structuring and problem modeling, which again represent more abstracted quantities.

One of the results of this difference, as mentioned previously, is the approach toward development of a program. In the actual creation of a program, experts tend to apply a top-down approach while novices tend to work bottom-up. It was also noted that as they evolve, novices tend more toward using a top-down development process [51]. For purposes of driving the novice toward expertise, this knowledge can be valuable. This is especially helpful when it is known that in spite of the expert/novice difference, the fundamental assumption related to human working memory capacity -- and discussed earlier in this chapter -- is that it is not different between experts and novices [22].

One of the last considerations for expert vs. novice discussion is the simple act of reading code. Perkins and Martin [48] found through interviews that while novices would commonly be able to write a segment of code, they did not commonly work through the code in their heads, a process called *desk-checking* by some [1], and *close tracking* by these authors. Very likely from experience, experts will commonly run a brief test of code segments to verify that they work within the appropriate constraints or limitations, but novices do not have the experience to drive this activity. In fact, it is common for novices to write segments of code and then resort to trial and error actions rather than trying to understand the program actions at the present level of scale or above [2]. It is also argued that this is further evidence of the narrow perspective held by novices.

To be clear, the issues found in this research do not by themselves oppose the teaching of introductory students to program. It is knowledge of, recognition of, and

finally adaptation to, these issues that can lead to improving the educational process. For example, it is generally accepted that experts have a significant base of experiences, memories, and schemas that are organized and contextualized as opposed to maintaining a large list of individual memories (see for example [10]). Incoming computer programming students by definition do not have these characteristics. And as novices, they will not acquire the quantity and quality of experiences in the short period of time offered by one or two semesters of programming study. Thus the appropriate strategy, and the findings of this section, should not lead to an attempt to create experts, but to provide tools that use the abilities novices do have to build the steps or provide the scaffolding [63] toward their improvement in the direction of expertise.

2.6 Hypothesis-Testing, Experimentation, and Hacking

While the expert/novice topic is an important combination of both the researched background and the practical foreground of programming, the last topic addressed in this chapter will lean more in the direction of the practical. There are parts of learning introductory programming that naturally require management as a result of students' natural desire to experiment or play with the tools available. Experimentation is not bad in and of itself. Indeed, as mentioned previously, both experts and novices will generate hypotheses and in most cases test them. However, too much experimentation on the part of any programmer regardless of ability will require much more time than should be required whether or not the experiments work. This can lead to frustration, incomplete project execution, or both, on the part of the beginning programmer.

Kirkwood identified "failing to plan or reflect, rushing straight to computers, constant floundering, and using random trial-and-error to debug programs" as evidence of "undesirable characteristics" [36] of student programmers. Rist [51] pointed out that experts do move into detailed coding operations during program design sessions, and they move easily between these modalities as needed for component design. He also noted that since novice programmers do not have access to the high-level design process, "they tend to flounder and search for a solution with little overall plan or organization" [51]. Again, the time used and the frustration generated by this process is not conducive to learning, or in some cases, to staying with the course.

In the course of evaluating the use of a number of programming languages by experts, Green, Bellamy, and Parker [32] found further evidence that programmers jump around to a certain extent when programming, and this tended to be related to the language. However, even in this study with fairly simple programs, the researchers were surprised to find that some of the programmers -- using the PASCAL language -- were using a stepwise refinement process (i.e., repeated passes through a program making improvements by degrees) that indicated global-view analysis and development of the program. And again, the experts in this case had the option to move between high-level and low-level programming activities.

In still another study, van Merrienboer [62] found that trying to compensate for measured impulsive and reflective student characteristics did not work, although when the students were given a choice as to the support they needed, they tended to fare better. Robins, Rountree, and Rountree [52] suggested that opportunistic exploration might be appropriate for novice students, but this was after pointing out the incremental

development and management of highly complex procedures that should be part of the process.

There certainly seems to be a place for experimentation and exploration in the programming process, and indeed it is certainly part of the creative endeavor that is programming. However, if this is the only strategy, unsupported by other more focused guidance, the evidence suggests that students will not experience as much success. Goldensen [29] pointed out that in many cases, procedural abstraction is put off for students until they get to more advanced courses. Perhaps the present state of introductory programming is evidence that this teaching strategy should be modified.

Robins, Rountree, and Rountree suggested that programming workbooks should teach and support "an explicit software development method to give some structure to the process" [52]. Soloway [57] suggested that in addition to learning stepwise refinement and planning skills, students should be taught a standard set of communication tools for discussing and understanding the programming process. Later in the same article, he pointed out that very few textbooks discuss appropriate programming organization, and that students are expected to pick up what he calls the "rules of programming discourse" [57] by observing examples of other programs.

The suggestions from research support a systematic format for learning to program, but this does not seem to be happening in the classroom. As mentioned earlier, programming is to a certain extent considered an artistic endeavor since there are so many different ways to solve any given problem. The unfortunate result of this tends to be that students experiment, they naturally generate short-term hypotheses, and then they follow their testing process, an activity commonly called "hacking". However, before

students can be allowed to try their own approaches, it would appear from the evidence that they should be provided a more secure foundation.

2.7 Chapter Conclusions

There are some human considerations related to learning to program, or to learning any new discipline, such as the limitations of memory and the potential for cognitive overload. There are also strategies that apply research to appropriate program text organization and the use of white space that can help. Moreover, using constructs such as the focal quantity that are recognizable and manageable are additional support. Finally, in order to fend off student "floundering", there is evidence for, and a call for, a structured and/or standardized programming process. The key to all of this knowledge as evidence, and as potentially actionable information, is that these programming and learning components are based in research and have a reasonable chance of supporting educational success. The next chapter will address more questions focused on the practical needs found in the research that will drive the proposed solution.

Chapter 3
Identified Needs for Teaching CS1

In the previous chapter, a review of the problems identified in introductory computer programming (CS1) courses, and researched over roughly the past 30 years was provided. While other characteristics may have been observed by the researcher or other CS1 educators, only those components with significantly supported research were provided. In this chapter, the same conditions are applied to specifically identified needs as related to the teaching of CS1. There are three primary needs identified and a small group of other needs identified in a fourth subsection.

3.1 The Need for Structured Programming

In their study of novice mistakes, Spohrer and Soloway stated, "we conclude that students are not given sufficient instruction in how to 'put the pieces together.' Focusing explicitly on specific strategies for carrying out the coordination and integration of the goals and plans that underlie program code may help to reverse this trend" [58]. The specific strategies will help students create both readable and operational program code. Davies [22] added to this with the idea that an incremental problem-solving process be used with individual program conditions and with frequent problem review.

Structured programming as its own term grew out of initial problems with disorganized program code [26] that was difficult to read and understand. It had its own name: "spaghetti code" [9] that was an appropriate descriptor as the program flow moved both forward and backward in the program with little organization. It was difficult for individual programmers to understand some of these kinds of programs [27]. The essence of structured programming was the difference between what were called *jump* programs that used the "goto" statement, which could lead to anywhere else in the program, or confined blocks of program code within *nested* conditional conditions (i.e., decision-making or branching areas).

However, structured programming as pertains to this thesis is not as formally related to its original concepts as it is a term used to provide students a standardized scheme with which to design and build a program. In fact, while the proposed program organization does follow the block-organized and nested structures suggested by structured programming, the emphasis herein is on using "some" structured process as opposed to a specifically identified "correct" process.

To begin with, the structure proposed includes a systematic design process prior to actual coding. Perkins, *et al.* [47] pointed out that students do not synthesize their own programming plans for a variety of reasons, and the result is a lack of success. Davies [22] added that programming strategies may be more important than individual components of programming knowledge. Spohrer and Soloway [58] argued that many bugs, or programming failures, are introduced into programs as a result of poor design, a condition they termed "plan composition problems" [58]. Providing a formal structure for the students is a way to support their learning by giving them a concrete scaffolding step

to stand on when they are first developing their programs. While much more on program planning and design will be found in the next section, it is organized planning that drives the development of structure.

To use Soloway's [57] example, students can observe a geometric proof on the board, but it is a different thing to replicate it, or synthesize a comparable proof once the students are on their own. This is a rationale that drives forming the structure itself before the programming process begins. If the structure in its simplest form is applied before the content is added, students can incrementally move the structure toward the desired programming goal, as suggested by Davies [22].

The next step to supporting the learning of structure is to systematically increase the complexity and depth of the program with repeated passes over the original structure and adding small scale improvements or refinements, a process called *iterative* or *stepwise refinement*. This means that every pass through the evolving program is a small step toward the solution, but the critical point is that no pass or step is very complicated or difficult, and the process is systematically repeated which supports consistency and familiarity with the process early on. Soloway [57] proposed this process but pointed out the failures of instruction and textbooks in the process of how to break down the problems into sub problems. In addition, one of the unfortunate points about this process is that it is dynamic and transitional; it is difficult to grade the steps and it can be difficult to see how the steps and the design went together in the final product [11].

Returning to the concept of advanced design, it is known that experts tend to retain program structures as mental models or schemas [52], but it is also known that novices do not have the schema libraries to hold these kinds of things, although they

move in that direction during this process. In addition, Robins, Rountree, and Rountree note that these mental models are hierarchical, as might be intuitively expected. This requires cognitive management and organization that novices are known not to have; however if this limitation could be overcome, the design and refinement process could be successful.

The next arguments for structure follow the research related to student perception of text organization. The block structure of what are called "nested" programs (i.e., programs with confined blocks of conditionally selected code as opposed to programs with goto statements) reduces the potential problems that Green [30] called "shopping", "treasure hunting", or "negation". The structure of the program helps organize the processing in a hierarchical way, which can lead to the structures students need for cognitive organization. Furthermore, as Gilmore pointed out the structural organization on the page or computer screen "are an effective means of supplying access to information which is not immediately apparent from the programming language" [27]. As mentioned in the previous chapter, this offers extra help to students who are trying to organize the programming structures into their mental models, and working memories.

It is not just the working mental models that can be improved by effective program structure. Merrill proposed a series of "First Principles of Instruction" which included research showing that not only is the local learning improved by structural organization, "problem solving (far transfer) is promoted when the structural features are carefully identified and explicitly mapped for the student" [40]. Again, the introductory student does not arrive in class with these abilities, but if a way can be found to support this, it will clearly be helpful.

One more strategy for teaching structure was proposed by Sweller, van Merrienboer, and Paas [59] based on other research that showed that worked examples could also provide both learning and guidance as well as models of structure. Worked examples by themselves may not be as effective; however, using the process of fading, which means starting with fully worked examples and working with the students away from the answers and toward their own programming knowledge [42]. Forward fading includes leaving out parts of the beginning of a problem solution and backward fading includes leaving out ending parts. Renkl, *et al.* [49] found that the backward fading worked better perhaps because students were given help starting and then could continue generating solutions even as the supporting components were being removed from the end of the problem-solving process. While the use of worked problems is not directly involved with the process of programming, it is certainly a tool that could be used by students who are still learning the structures necessary for design and development.

Program structure has the benefit of being both operationally and pedagogically appropriate for teaching introductory programming. Poorly structured programs, including the ones that might work, are not the desired product of successful students in a CS1 course. As mentioned, they may not represent good design and planning, and they may be difficult to read and understand, and later to debug or upgrade if they are not structured well [27]. In addition however, providing structure, organization, and clear guidance [40] to students is known to be effective. In one form or another, structure must be part of the teaching solution.

3.2 The Need for Planning and Strategy

As mentioned previously, it seems to be natural for students to want to get straight to the programming process when they are given a problem to solve [36]. However, in the conclusion of their comprehensive review, Robins, Rountree, and Rountree suggested "that the most significant differences between effective and ineffective novices relate to strategies rather than knowledge" [52]. It is interesting to note that this is a comment related directly to the learners, but it could also be directly associated to programs themselves [58].

Programs can be written without planning if the problems are trivial, but they are unlikely to be successful for non-trivial or more complex problems. One of the first actions that must occur is that the programmer must specify a detailed plan which must then be decomposed [56]. This would seem to be good advice for anyone tackling any large-scale project, but as mentioned in the previous chapter, even the student who stops to create the plan will have trouble maintaining the plan in working memory. In the course of beginning the programming tasks, some or all of the parts of any plan will be lost unless they are "stored" in the text of a stated plan [22], or in some other way while the programming tasks are conducted.

Spohrer and Soloway [58] argued that goals and plans should be specified for students, in addition to other explicitly specified design actions. Rist [50] discussed top-down planning as the global strategy and bottom-up development as separated from the original problem structure, but more manageable given working memory limitations. It would appear that both of these strategies are necessary, although they need to be

managed. Top-down planning gives the programmer the abstracted "birds-eye view" of the program; this is critical in order to identify the major steps needed to solve the original problem. However in an attempt to model the comprehension of a programming solution, Brooks [12] identified stages between the problem domain and the programming domain. It appears to be a key point that there must be at least some delineation between the problem and the program for a successful problem-solving/programming process to be conducted.

The alternate offered by Rist [50] is the bottom-up strategy which he stated, "separates the solution structure from the problem structure" [50]. There appears to be a natural but complementary division here that shows the value and the usefulness of both top-down and bottom-up strategies. The top-down strategy maps the overlying solution to the problem, and the bottom-up strategy solves the little problems driven by the overlying strategy.

Certainly in the course of devising a plan, some amount of working backwards from the goal must be conducted, as Rist pointed out [51]. On the other hand, once the needs of the program are defined, an abstracted top-down overview of the program should be devised that provides "overarching supportive information" first, but then "procedural information should be presented only at the particular point where it is required", as stated by Paas, Renkl, and Sweller [42]. It should also be noted that in a later paper, Rist [50] discussed the process of working from the abstracted quantities to the concrete ones for both the actions of program design and program understanding.

Merrill [40] offered a generalized four-stage approach to problem solving, and to teaching problem solving that begins with the problem statement and ends with the

actions that conduct the process, with an abstract-to-concrete evolution in between. Enough evidence has been discovered to support this general strategy, although some of the same people who called for this also noted that novices may not have the remembered schemas for thinking at this higher level [51]. Nevertheless, Rist continued later in this paper to point out that plan-building methods could become a focus for teaching novices.

One key to resolving the higher-level to lower-level program development may simply be the writing process. In a research process that studied the teaching of higher-level thinking with program development, Kirkwood [36] found that the students who were developing quality programs viewed the possession or construction of a written design as important for several reasons associated with developing a good program and for minimizing programming problems (e.g., early elimination of problems, and later bug tracking).

With or without prior student knowledge of programming schemas, one way to minimize problems is to break them down into smaller quantities, which is obviously an extension of the top-down program development structure. van Merrienboer, Kirschner, and Kester [63] argued for teaching smaller or more simplified whole parts of a larger task in order to reduce cognitive load. This turns out to be a natural fit for programming because the top-down process breaks the larger program into smaller, modular components. Gerjets, Scheiter, and Catrambone [25] concurred arguing with their research that teaching and working with modules is superior to teaching groups of concepts or related concepts which they call a molar approach.

Catrambone [18] found that students were improved both in conducting mathematical operations and in understanding them by breaking them down into smaller

"sub goals", also noting that the use of abstract labels was less likely to lead to mistakes. Even when Ayres [5] found errors in sub goals, it was proposed to be due to cognitive overload due to the complexity of the sub goal, a condition called *stage effect*. These mathematics problems could not be broken down to a small enough level to reduce the cognitive load and to support successful problem-solving actions. This issue led Robins, Rountree, and Rountree to identify "the schema/plan as the most important building block of programming knowledge" [52] although they also noted that the schema/plan is an "ill-defined concept" [52].

Breaking a program into more easily understandable and less complex modules can lead to improvement in problem-solving performance for students with either low or high levels of previous knowledge [25]. This is both intuitively reasonable and empirically evident. The two issues that must be resolved in order for this knowledge to be valuable are: 1) how small does a module need to be in order not to overload learner cognition? and 2) how are the modules managed during the breaking down and reassembly process?

The answer to both of these questions can be driven by the stepwise refinement process mentioned earlier [57]. The problem is stated at the abstract level and then slowly but consistently expanded [51] toward smaller sub goals or modules until the modules are within the cognitive "reach" of the novice learner [25]. This process of expansion will in itself cost cognitive overhead; however students would not have to maintain the process in working memory, but instead can effectively "store" the levels and sub goals on the screen of the computer, or on paper [32]. Working through this process itself synthesizes a complex knowledge structure that can help structure

knowledge in working memory [25] with the caveat that the structure may not -- and in fact, probably will not -- be internalized at first. Nevertheless, the evolution of this activity will naturally go through the fading process that is known to be effective with learning complex tasks [42]. The planning and design process is by nature complex; however, once this is recognized and addressed, there is potential for working through it.

3.3 The Need for Cognitive Considerations

One thing that many people might not consider teaching in an introductory computer programming course would be cognitive considerations. By itself, this is unfortunate because teaching cognitive and metacognitive strategies have been shown to improve learning [10]. In the particular case of designing a program, which is really just a formalized process for solving a problem, it would seem even more important. Mayer, Dyck, and Vilberg's [38] research concluded that while the evidence is weak in relation to general intellectual skills, focusing on specific cognitive skills that are related to programming could be more fruitful.

Soloway [57] pointed out that knowledge and strategies must be explicitly taught in order to get to the higher-order or more transferable problem-solving strategies. Stimulating mental models that may support or encompass the new learning is important [40], and although it is known that introductory programming students have few if any pertinent programming mental models, they do have at least primitive schemas related to solving problems and step-by-step operations. These students can start with the simplest of approaches to developing the program and once they have a small scale model, this can be manipulated and expanded [51].

Beyond directly introducing cognitive characteristics, the process of reflection and review can help [40]. In programming, this can be accomplished at two levels. Reviewing the program as the large-scale solution actions can result in both an understanding of the program, and it can lead to identifying semantic or logical issues that might keep the program from solving the problem. In addition, reviewing segments of program code (i.e., working through the code in one's mind or on paper) can lead to identifying the smaller bugs and problems in the program, or verify that this particular module of the program works correctly.

Besides reviewing programs at various levels in ones mind, it is also suggested that they engage in self-explanations [19]. Again, the act of talking through a problem, whether out loud or not, supports the reflection and review process that supports cognitive improvement and therefore better learning. One other result of the review process is actually finding mistakes. As Merrill pointed out "Most learners learn from the errors they make," [40] and continued to note that the learning is strengthened when they are given error finding, error correcting, and error avoidance strategies.

Another somewhat intuitive condition is that abstractions cannot commonly be learned directly, they must be introduced and practiced as smaller, more atomic components [3]. This can occur during a stepwise refinement process, but while it may not be immediately obvious, the program must be written for both the programmer and the computer [57]. Obviously the computer must be provided the correct instructions in order to accomplish its tasks, but the programmer "needs to have an explanation as to why the program solves the given problem" [57]. Even if novice programmers can

understand the process, if they are struggling to understand it due to the clarity of the program, their cognitive load will be increased.

A final cognitive consideration is that of cognitive styles. Some correlations have been found between programming and field independence, high reflectivity, locus of control, and introversion, however not enough research has been conducted to find empirically sound relationships [8]. This supports the present thesis in that the more fundamental human characteristics such as working memory and cognitive load can be the focus of the research.

Cognitive components are an integral part of the programming process. Indeed, a computer program is a cognitive process, even if it is not a highly intelligent one. As a result, cognition has been strongly integrated into this thesis. However, where certain cognitive considerations were not integrated elsewhere, they have been presented in this topic. As expected, cognitive components will continue to contribute to parts of this thesis, including the next topic which will wrap up this "Needs" chapter with smaller contributors to the identified needs for a teaching process.

3.4 Other Contributing Needs and Considerations

While the major needs previously identified in this chapter are supported by significant quantities of research, the final group of components are of a smaller scale. Nevertheless, each of them was found to contribute considerations for the effective education of introductory programming students. Thus, these will be presented in this section.

The first small group of considerations would commonly be good ideas in any classroom. However, as Robins, Rountree, and Rountree pointed out, providing "clearly stated course goals and objectives, stimulating the students' interest and involvement with the course, actively engaging students with the course material, and appropriate assessment and feedback" [52] do lead to a student-centered and effective learning environment [10, 23]. These teaching components must play a part in any educational tool being considered.

The next consideration is also a generally accepted one, with some clarifying conditions. Practice is obviously a need, but Merrill [40] pointed out that consistent focus on the learning goal as well as consistent and appropriate feedback are important contributors to the student learning experience. Sweller, van Merrienboer, and Paas [59] showed several findings that while simple repetition may not be helpful, varying practice conditions demonstrates improvement in students. They noted that this may seem inconsistent with the possibility of increased cognitive load, but hypothesized that as long as the focus was on the appropriate learning outcome, cognitive load was not significantly increased.

As previously mentioned, there is evidence that worked out problems might be of help toward the learning; however while commenting that both tools were helpful, Bunch [15] noted that there was little difference in student feedback between worked out problems and progressive practice which includes working with increasingly complex scenarios. Renkl, *et al.* [49] argued that worked out problems are no longer preferred for supporting what they called automatic performance, and what would be called synthesizing a program in this thesis. Then again, with reference to the other research,

they also noted that worked out problems are preferred by novices and can be more effective if they are managed appropriately with components such as fading, as mentioned previously in this chapter.

The problem/scenario learning environment must also be a consideration. Generally speaking, all the research supported whole program learning with some variability. For example, van Merrienboer, Kirschner, and Kester [63] used whole-task operations but tended to start with low intensity sub goals, to mitigate problems with cognitive load. Merrill also used a problem-centered, or what he called "real-world activities" [40] approach, but did not directly address the cognitive load issue. Robins, Rountree, and Rountree [52] supported the case-based or problem based learning activities as well.

Interestingly, all the cognitive studies reviewed showed a propensity toward programming whole-task conditions (i.e., whole programs) while there were no cognitive studies found that reported on the so-called "programming in the large" teaching strategy that has come into being in recent years (see for example [20]). While this was not the focus of this research, the programming in the large paradigm seems driven by the large-scale characteristics of present-day programs in industry, while the researchers -- and especially cognitive researchers -- continue to study what is sometimes called "programming in the small" which is focused more on the achievements of students on whole programming products.

There are other tools that can support novice programmers including as examples: computer tutorials [24], graphical tools such as RAPTOR [16], and interactive environments such as Alice [54]. These are all tools designed with the interaction and

engagement specified previously in this section; however, with small exceptions, they tend to be line-by-line programming interactions. RAPTOR and Alice allow for the use of subroutines, and RAPTOR's graphical presentation allows for a kind of graphical abstraction using boxes that represent sub goal modules.

As a tool, RAPTOR has the capacity to be developed in a stepwise refinement process, developing the large scale program and then developing sub goal/module components in iterative passes. RAPTOR was developed for non-CS major students and has been successful with this audience [16, 17]. Students can also translate from the flow chart organization of this tool to their own program development; unfortunately this adds steps and time that are both precious to introductory programming students.

There are certainly more identifiable needs for the effective teaching of introductory computer programming, but the ones provided in this section were the ones found in the research and can be empirically substantiated. This section did allow for the identification and elucidation of individual components that did not fit under the umbrella of the previous sections in this chapter, so the research could be as thorough as possible.

3.5 Chapter Conclusions

As this chapter has shown, there are specific needs of students learning to program computers, and of the educators attempting to accomplish this feat. The characteristic that makes these needs synergistic is that they parallel each other. The development of reasonable, non-trivial programs requires appropriate structure; any other format will become ineffective and/or unusable before its life cycle should be completed. Indeed, this may even occur before the program is completed. In addition however, the

teaching of students in virtually any discipline of substance also requires appropriate organization, structure, and guidance.

The same argument aligns with the need for planning and strategy. Learning must be strategic, and programming must be strategic. With appropriate planning and strategies, both can be successful. Even cognition works in parallel. Student cognition will lead to development of the "cognitive" abilities of the program that will solve specified problems.

This makes the teaching of introductory computer programming somewhat unique. And arguably, it offers unique opportunities to improve this educational task, and to contribute to the educators who attempt it. Having provided the background in this and the previous chapters, a proposal will be made in the next chapter based on this research, to develop a learning process for developing effective programs along with successful learners.

Chapter 4

Proposal: The Five Step Programming Process

Perkins and Martin wrote, "Rather than expecting programming instruction of itself to boost cognitive strategies, one should teach cognitive strategies as part of better programming instruction" [48]. It is not only intuitively appropriate, but the concept of teaching strategies that support both cognitive and programming improvement are supported by the research, which also states that just teaching programming has not been linked to cognitive improvement to any significant extent [38].

This chapter will introduce a proposed learning process based on the research reported in the previous chapters. The three sections of this chapter will be as responsive to and as guided by the research, as possible. Components of the process that are not directly supported by the research will be primarily informed by logistical needs and local classroom conditions.

4.1 The Proposed Procedure

The procedure is broken into five parts or steps. Part of the process of knowing where to break the parts is supported by the research, and part of it is related to the functional operation of creating a program. For example, the first step is purposely very simple and requires very little time while the cognitive involvement and the time required

is larger for each succeeding step thereafter. The steps will be presented in order and with supporting information.

4.1.1 Step 1 - Creating a simple high-level solution

The first step starts with a previously specified "complete program" format provided to the students. This format includes: 1) one or more header files, 2) zero or more global constants, 3) zero or more function prototypes, 4) exactly one **`main`** function, including an appropriate return statement, and 5) zero or more "supporting" function implementations (i.e., functions with the program code written as needed and/or specified). All functions other than the **`main`** function are considered supporting functions for purposes of this standard. An example of this program format is provided in Figure 4.1.

```
// Header files
#include <iostream>

using namespace std;

// Global Constants
   // none

// Function Prototypes
   // none

// Main Program
int main()
   {

    return 0;
   }

// Supporting Function Implementations
   // none
```

Figure 4.1. Standardized Program Format

Starting from the base program format or template, the students are directed to write the five to seven main actions of the program in the form of comments, meaning in English text that is not processed by the compiler. These main actions are simple statements that will drive the remainder of the program and in fact become the abstractions that define the overall program. Still, as simple statements provided at the start, they do not require students to be competent with or fully knowledgeable of the concept of abstractions at this point.

The key points to development of this step are as follows. First, just by writing the major steps, the students are developing the recommended plans or goals, even if they are in a very simple format. Second, by writing a few statements in their own language, as opposed to the programming language being learned, students are separating the problem from the program right from the start. Third, by limiting the number of statements to no more than eight to ten, the working memory storage itself is not significantly taxed. It should be noted that this number of items is greater than the three to four to seven chunks identified as limits in the research, however since these items are written down into the text file, the external "storage" process also discussed in the research is supported.

Finally, this step is a simple beginning to an abstracted top-down process that will guide the students to synthesizing a programming solution to the specified problem. The concepts of both abstraction and top-down strategies will have been discussed with the students prior to this point; however, these concepts do not have to be involved in the actual problem-solving process. This is an important condition as the research points out the problems with increased cognitive load in novices due to maintaining the rules of structure in working memory while actually trying to solve the problem.

Figure 4.2 shows the basic structure of a program that calculates the roots of a quadratic equation. Note that in order to minimize distractions, only the main part of the program will be shown until other parts of the program become pertinent. Also note that the text is organized with indenting and is separated for easy visibility. This is again from the research related to white space and text organization. Finally, to repeat, the important point about this step is that it is very simple and does not appear threatening to students who are not familiar with the programming process. This is specifically designed to address the issues of frustration and potentially weak problem-solving skills with which the students may be starting.

```
// Main Program
int main()
   {
    // initialize program/function

       // initialize variables

       // show title

    // input coefficients

    // process the data

    // display roots

    // shut down program
    return 0;
   }
```

Figure 4.2. First Step of Programming Process

4.1.2 Step 2 - Expanding on the original solution

Having begun the problem-solving process with the first step, students recognize that they have not moved very far into writing the program; however, they also recognize that they have taken a first step, and with presumably some sense of accomplishment. The second step of the process requires the students to expand on the individual statements they have already written. At this point, students have been given a "starting cue", and as the research calls for "direction", the first step, and subsequent steps, provides this in an ongoing fashion. For the most part after the first step, no large scale synthesis is required of the students since each step will be guidance as to the next sub goal steps. Cognitive load should never increase significantly, and at the same time, the library of schemas held by experts -- which is known not to be available to novices -- is not required in the guidance process. The expanded statements for the second step of the example program are presented in Figure 4.3.

Examples of expansion on the first step statements are to show input of the three coefficients under the statement "input coefficients". A more expanded example is the specification to calculate the discriminant, then calculate its square root, then calculate the denominator, and finally calculate the roots of the equation which are sub goals of the "process the data" statement. These are four steps that do require the student to think through the quadratic equation seeking small steps that will solve it.

The increased value of this process is that students are guided to keep the steps small, which will keep the cognitive load down, but by its nature, they will also be increasing the modularity of the program, which is a desired part of quality program

structure. Note also the use of both white space and the indented structure from the research that provides students with a hierarchical structure that will support their understanding of the top-down structure as it evolves.

```
// Main Program
int main()
   {
   // initialize program/function

       // initialize variables

       // show title

    // input coefficients
       // input coef a, b, c

    // process the data
       // calculate the discriminant

       // calculate the square root of disc

       // calculate the denominator

       // calculate roots 1, 2

    // display roots
       // display root 1

       // display root 2

    // shut down program

       // hold screen for user

       // return 0
    return 0;
   }
```

Figure 4.3. Second Step of Programming Process

The "display roots" step is broken down into the obvious two steps required of the two roots, and the "shut down program" step is expanded to include holding the screen for the user to view it before the program stops. A key point related to separating the problem from the program is that when step two is completed, students should be able to

"run" the program in their minds in an attempt to verify that their program conducts the appropriate operations. They should be able to identify how the program will, or will not, work and they should be able to understand if an important part of the process is missing. The key to this is that the students are still doing this in a language they understand, again without the cognitive load of trying to remember what certain programming statements would, or would not, do.

It should also be noted that even though the students are fully involved in a top-down development process, they do not have to keep the management, or even significant awareness, of this process in their working memory in order to continue forward.

4.1.3 Step 3 - Identifying and specifying program modules

At this point, students have made one small initial step with a few lines of commented text, and then they have expanded each of those lines where appropriate with a few lines of text related to the sub goals needed. The second step was slightly more complicated than the first, especially since students should have really been trying to verify the correct operation of their program. At this point in step three, little more is done to actually solve the problem, although this could occur if students notice a part of the solution that they had previously missed. Instead, implementing step three moves the students into expansion of the modular components into which their program has been broken.

For example, where the program calls for showing the title, the tool to be used for this operation would be the combination of the **`cout`** object and the insertion operator (i.e., **`<<`**). Again with consideration for cognitive load, the student is only required to

consider one commented program step at a time, but they must also consider which tool is appropriate for implementing this step. Figure 4.4 shows how the students indicate the use of the tool.

```
// show title
   // function: cout, insertion (<<)
```

Figure 4.4. Indication of the tool(s) to be used

According to the research, there should be room in working memory for this to occur. However, not all the modules will be tools or subroutines -- called *functions* in the C++ programming language -- that have been previously created. Students will need to synthesize their own subroutines for some of the actions unique to their particular program.

For example, this particular program will need to prompt the user for each of the three coefficients, acquire each of them, and return them to the program for processing. Since this is a task unique to this program, students should recognize that they must create the tool themselves. At this point, they must conduct two tasks: They must indicate the use of the tool, as shown in Figure 4.5, and then they must specify information about the function, which will indicate its characteristics and actions.

```
// input coefficients
// input coef a, b, c
   // function: getCoef
```

Figure 4.5. Indication of the student-generated tool to be used

The specification part of this process occurs above in the program's "Function Prototypes" area. This part of the process will again elevate the cognitive challenge to students, although it continues to be guided and structured to minimize cognitive overload and/or frustration. There are five parts to the specification. The first is the function *name* - this should be one or more combined words starting with a verb to indicate action; **`getCoef`** indicates that the function will "get" or acquire data from the user. The second specification part is the function *input* - students must think about what the function "needs to know" to accomplish its task, and they must think about what data type would be appropriate for the input data. In this case, since each coefficient must be prompted with a different message, a string prompt is needed.

The third specification part is the function *output* - the function may return output back to the calling function (i.e., the main program) as this function in fact does by returning the acquired user input, but it could also output data to the screen or to a file. Again, students should indicate what is output and what data types might be involved in the process. The fourth specification part are the function *dependencies*. Many times functions must rely on other functions to accomplish their goals. For example, the **`getCoef`** function will be prompting the user for input and capturing it, so it will need to use some input/output (I/O) tools in the iostream library. Finally, the fifth specification part is the function *process* - what is it that the function will be doing, written in two or three lines of text at most; this function will prompt the user, acquire input from the user, and then return the input value to the calling function. An example of the format for the specification component is shown in Figure 4.6.

The specification process is somewhat larger scale than the one line at a time process conducted down in the main program. Nevertheless, it is still a stepwise process with specific actions associated to a specific set of given terms. The students still only have to address the name issue, and then the input issue, and then the output issue, and so on, one item at a time. Again, the appearance of following a series of steps is less daunting, and again, the cognitive load is managed.

```
/*
Name: getCoef
Input: prompt string (string)
Output - returned: coefficient value (int)
Dependencies: cout, cin
Process: prompt user for coef, get coef,
         return coef
*/
```

Figure 4.6. Specification of a student-generated function

As was found in the research, it is desirable to implement iterative or stepwise refinement in order to develop a program. As was also noted, this has the functional and pedagogical value of slicing the program development process into smaller, more cognitively appropriate, chunks. Step three continues to be conducted this way with students looking at the next required step of their program, making a decision on the tool(s) to be used, and then specifying their own tools (i.e., functions) as needed. This is literally a top-down process on their source code text page, and continues to be carefully managed in terms of working memory and cognitive load demands. A sample of other function specifications is shown in Figure 4.7, and a continued sample of the program development is shown in Figure 4.8.

```
/*
Name: calcDisc
Input: three coefficients (int)
Output - returned: discriminant (double)
Dependencies: none
Process: calculate discriminant and return
*/

/*
Name: calcDenom
Input: coefficient a (int)
Output - returned: denominator (double)
Dependencies: none
Process: calculate denominator and return
*/

/*
Name: calcRoot
Input: denominator, discriminant (double),
coefficient b (int)
Output - returned: root (double)
Dependencies: none
Process: calculate root and return
*/
```

Figure 4.7. Further examples of function specifications

```
    // process the data
       // calculate the discriminant
          // function: calcDisc

       // calculate the square root of disc
          // function: sqrt

       // calc the denominator
          // function: calcDenom

       // calculate roots 1, 2
          // function: calcRoot
```

Figure 4.8. Continuation of the program function development

4.1.4 Step 4 - Developing the skeleton program

Each step in this process moves in the direction from the problem solution to the programming solution. Step one was the simple outline of the program, step two was the expanded outline, but still all in text. Step three continued to be developed in text, but this was the transition step where program code was conceptualized, even if it was not actually written. Step four begins the process of actually writing the program code. The process will continue with the stepwise refinement format, but once again, the "next steps" required of students are guided by the results of all the previous steps. For example, where the program was previously written in text with the tools specified, the actions are now put into play in the main program. Figure 4.9 shows the evolution of the example shown previously in Figure 4.8.

```
// process the data
   // calc the discriminant
      // function: calcDisc
      disc = calcDisc( coefA, coefB, coefC );

   // calculate the square root of disc
      // function: sqrt
      discRoot = sqrt( disc );

   // calculate the denominator
      // function: calcDenom
      denom = calcDenom( coefA );

   // calculate roots 1, 2
      // function: calcRoot
      root1 = calcRoot( denom, discRoot, coefB );
      root2 = calcRoot( denom, -discRoot, coefB );
```

Figure 4.9. Program code provided for processing operations

Note that each line of code is guided by the comments provided, and the appropriate tool (i.e., function) is used at the appropriate location. Also note the continuation of the white space and structural organization of the comment text that is now being translated into program code. The other two components that must accompany each line of code that contains a new function are the function prototype that will now be placed in the "Function Prototypes" area immediately under the step three specifications at the beginning of the source code file, and the now stubbed out functions that are placed in the "Supporting Function Implementations" area below.

There is a critical order for this process that incorporates both the iterative top-down process and the functional need to write a program without syntax errors. Thus, the following protocol must be conducted by the students. First, they write appropriate program code in the **`main`** function starting at the top. When they arrive at a location that requires one of the student-generated functions, they must first go to the "Function Prototypes" area of the program under the location of the function specifications and create a function prototype.

The prototype has a standardized form that will be driven by the specifications generated in step three. The return (i.e., output) value is placed as specified, the name of the function is placed as specified, and then the parameters (i.e., the input value(s)) is/are placed as specified. Once again, this is small step cognitively because all of the requirements for the prototype have already been specified. This is primarily a translation process. At this point, the students must compile the program, which will work correctly if there are no syntax errors. An example prototype is shown in Figure 4.10.

```
/*
Name: getCoef
Input: prompt string (string)
Output - returned: coefficient value (int)
Dependencies: cout, cin
Process: prompt user for coef, get coef,
         return coef
*/
int getCoef( const string &prompt );
```

Figure 4.10. Function prototype under the specifications area

The key to keeping this process cognitively managed is that if students do make a mistake with creating the prototype, they only have one line of code to resolve. While most compilers may provide cryptic error messages, students are not overloaded by the possibility of problems elsewhere in the program. They have only to resolve issues that occurred in the one line of code.

The next step continues this support as well as the cognitive management. Students must now create an empty function, called a *stub function*, down in the "Supplemental Function Implementation" area. This involves copying the prototype exactly as it is, pasting it in the area below, creating open and closed curly braces for the function, and finally, if the function requires a return value, placing a dummy return value in the function code block.

Creating the stub function now requires placing at most two lines of code in addition to the two curly braces in the "Supplemental Function Implementation" area. Once again, if any syntax errors result from this process, it is clear where the errors are,

and where the corrective actions must be focused. An example stub function is shown in Figure 4.11.

```
int getCoef( const string &prompt )
   {

    return 0; // temporary stub return
   }
```

Figure 4.11. Stub function example

The final step for installing this function into the program is placing it in the program code as shown previously in Figure 4.9, and again in Figure 4.12. Again, the student places one line of code that includes the function operation, then compiles, and then resolves any issues that may have arisen. The focus of the learning is on this one step, and the cognitive load is not overloaded if an error is introduced at this point.

```
// input coefficients
   // input coef a, b, c
      // function: getCoef
      coefA = getCoef( "Enter coefficient A: " );
      coefB = getCoef( "Enter coefficient B: " );
      coefC = getCoef( "Enter coefficient C: " );
```

Figure 4.12. Usage of student-generated function in the program

When students are finished with step four, they have a fully written main program. If they followed through with their original design from steps one and two, and if their functional operations from steps three and four are consistent with their plans, this function should work. It will not work correctly at this point because the supporting functions are not operational. However, students should have a reasonable feeling of

confidence that they do not have to make any further modifications to this part of the program. As discussed in the research, there are places in programs where some experimentation may need to occur and this could happen in the **`main`** function when the program is completed.

One benefit of this process is that if a semantic or logical error is discovered at the end of the programming process, the program is already broken into modules so that the overlying process can be understood and errors resolved. Again, this is both effective programming and pedagogically appropriate since students have worked their way through a top-down process, and should have a reasonable understanding of how their program works. One of the important points about understanding the program is that as long as they created the text comments and developed the program with clear steps, they are not required to hold the whole program in working memory, which the research says they cannot do. They should be able to review their program steps in a way that does not lead to cognitive overload, and yet still leads them in small steps to how the program works.

To continue with this "small step" strategy, the last component of completing step four is to write brief comments into each of the stub functions. These are essentially step two operational descriptions of the functions, which are by nature, already somewhat small operations since they are sub goals of the main process (i.e., the program). Writing these comments allows the students to think ahead about their next steps; however since they do not write code at this point, the number of steps and the complexity of the operations remains simple. An example of the required comments in the stub function is provided in Figure 4.13.

```
int getCoef( const string &prompt )
   {
    // initialize function/variables

    // display prompt to user

    // acquire user input

    // return acquired value
    return 0; // temporary stub return
   }
```

Figure 4.13. Creation of descriptive comments in stub functions

When the students have had some practice with writing these simple comments on several assignments, they are then required to develop one level of the functions in the step three format which includes identifying and then specifying functions that will be required for only the functions called directly by the **`main`** function/program. Toward the end of the semester, students will be expected to write step three comments, function identification, and function specifications for almost all of the functions that will ultimately be required by the program. While it is expected that the students will have developed a level of competence at the end of the semester, the stepwise process continues to only require attention to small quantities for tasks involving each module.

4.1.5 Step 5 - Completing the program

The final step of the process effectively applies a bottom-up strategy even though the program itself has been written. Students are now required to implement (i.e., write the code for) each of their function modules. If the program has been well-developed to this point, the function/modules will address one programming task and will be relatively easy to code. This is especially true since they should have a reasonable set of

instructions provided for them to follow as a result of the last part of step four. An example of the implementation of a function is shown in Figure 4.14.

```
int getCoef( const string &prompt )
   {
    // initialize function/variables
    int userInput;

    // display prompt to user
    cout << prompt;

    // acquire user input
    cin >> userInput;

    // return acquired value
    return userInput;
   }
```

Figure 4.14. Creation of descriptive comments in stub functions

Again, this strategy asks the students to make small forward steps with each module, but because the "memory" of the program evolution is "stored" in the text of their program, their cognitive load consists of following their instructions and solving smaller problems. One of the strategies continued from the step four process is to make sure that they compile their programs after every line or two of writing code. This again makes the error-resolution process very narrow, requiring minimum cognitive load, and supports a continuation of the stepwise process through the coding activities.

At this point, the program should run correctly, and in many cases, when this procedure is followed, it does. However, under those circumstances where a semantic or logical error has been introduced into either the main function or one of the supporting functions, the process of diagnosing and resolving is made significantly easier as a result of the program modularity. Students can view the output and identify problems that may

have occurred in the input or output operations, or in the mathematical processing. If viewing the program's displayed operations is not informative, students can still trace through the program in a variety of ways that will continue to be aided by the integrated program modularity.

4.2 Reviewing the process

Managing cognitive overload is the primary focus of the Five Step Programming Process, and it is managed in several ways. First, the problem and the programming are kept separate for as long as possible, while students actually design the solution in a stepwise format but without concern for program code. Next, by the time steps one and two are complete, there is little need to tax the working memory for more than what it takes to resolve each previously written step. Students can apply their limited working memory tools to the task, but at the same time, they don't have the immediately at-hand experience memories to which experts have access. However, students can call upon the small-scale problem-solving abilities that they do have and/or are being trained with, and resolve small issues one at a time.

Another important part of the process is the use of top-down stepwise refinement. This is not just an effective way to write a program. It also offers the students both focused and varied task repetition. For example, step one calls for one primary pass through the commented statements to solve the problem in a very general way. Step two calls for a second pass that expands on the individual commented steps of step one. Once the commented steps have been written in step two, students should take another pass or

two through these steps to make sure that the program will work as expected when following their instructions.

Step three continues with multiple stage-like steps. First students can go through identifying where the module/functions are needed, and having completed that pass, they can go back through and implement the individual function specification process. As a note, the creation of specifications can be incorporated into the module/function needs pass, but this only changes the number and length of the varied repetition experienced by the students.

In step four, the development of the **`main`** function program code is one large-scale pass while the integration of using, prototyping, and stubbing the functions are again, small but consistent passes. There is one more pass made through the stub functions to provide the descriptive comments. Even after the development of one program, students will have established an experience base of some depth, even though the items passed through are not significantly deep or complex.

Step five wraps up the iterative process by making one more pass through the stub functions to fill out the code. The master pass is over the individual functions, but the "sub goal" passes occur within the functions. To reiterate, the process is systematic and consistent, and there are several stepwise refinement passes throughout the process whose operational structure is the same, but whose contents vary almost every time. Having worked through a small number of full programs this way, students will have begun the process of developing schemas in an orderly way, and in a way that does not overload cognition or working memory. This is the goal.

4.3 Evaluating the Process

The goal of this research was to begin the process of both evaluating the Five Step Programming Process and refining it where needed. The first step in the direction of conducting this evaluation was a pilot study to observe how the students felt about the process, what suggestions they might have had for refining it, and how it might have helped them learn to program.

The process was taught to the students in the stages mentioned in the previous sections across most of a semester. Near the end of the semester in the twelfth of fifteen weeks, the students were provided the opportunity to respond to an anonymous survey. They were not required to do this, but they were offered food gift card incentives for their participation. The survey itself was online, anonymous, and administered by the thesis Advisor who was not involved with the teaching of the course. Nine questions including one open-ended question were provided in the survey. These are provided as Appendix A of this thesis in the completed form which includes the IRB consent information and decision-making question. Further explanation of the process and the results are provided in the next chapter.

Chapter 5
Implementation and Results

5.1 Implementation overview

The research was conducted using an online anonymous survey tool during a one week period starting on a Monday morning at 8:00 a.m. and ending on the following Monday morning at the same time. Students were formally recruited by an Institutional Review Board (IRB) trained staff member who read a scripted informational document to the two introductory programming classes in the Computer Science and Engineering Department during the beginning of each class during the one week period.

The nine survey questions described in the previous chapter and also provided in Appendix I of this thesis provided some opportunities for simple quantitative information using multiple-choice questions which included how much they used the Five Step Programming Processes (hereinafter called the "process"), how helpful it was in their development process, how it may have helped with the time taken to develop their programs, and how much the process may have helped with learning to program. The data was analyzed and is reported in this chapter.

The remaining questions offered students the opportunity to respond in their own words to questions related to difficulties they may have had with the process, the

potential applicability of using this kind of process in other courses, and ways the process might be condensed down or expanded out. In addition, an open-ended question was asked of the students offering them the opportunity to bring up any other issues they might have related to their experience with the process. This data was organized generally by the focus of the various responses and is also reported in this chapter. Examples of each grouped response are provided in this chapter, and the entire data set is provided as Appendix II.

Twenty-eight students signed into the survey system, but only twenty-six students completed most or all of the questions. While this response represents 1/3 of the 78 students who could have taken it -- which is a good response rate -- the number of students was not quite high enough to support significance in any of the quantitative analyses. This will be reported in the next section of this thesis along with a discussion of the other findings of these questions.

The remainder of this chapter is divided into two main sections. The first section reports on the quantitative data collected, and the second section provides analysis and reporting on the written student feedback. Brief concluding remarks related to the data collection and analysis will be provided at the end of the chapter.

5.2 Quantitative Questions

This section will discuss and report on the data collected on the four multiple-choice questions provided in the survey.

5.2.1 Usage Percentages

The first question asked was "How much do you think you used the Five Step Programming Process for working out your most recent laboratory program?". The question offered five choices between 0% and 100% in increments of 20%. The overall results are shown in Figure 5.1.

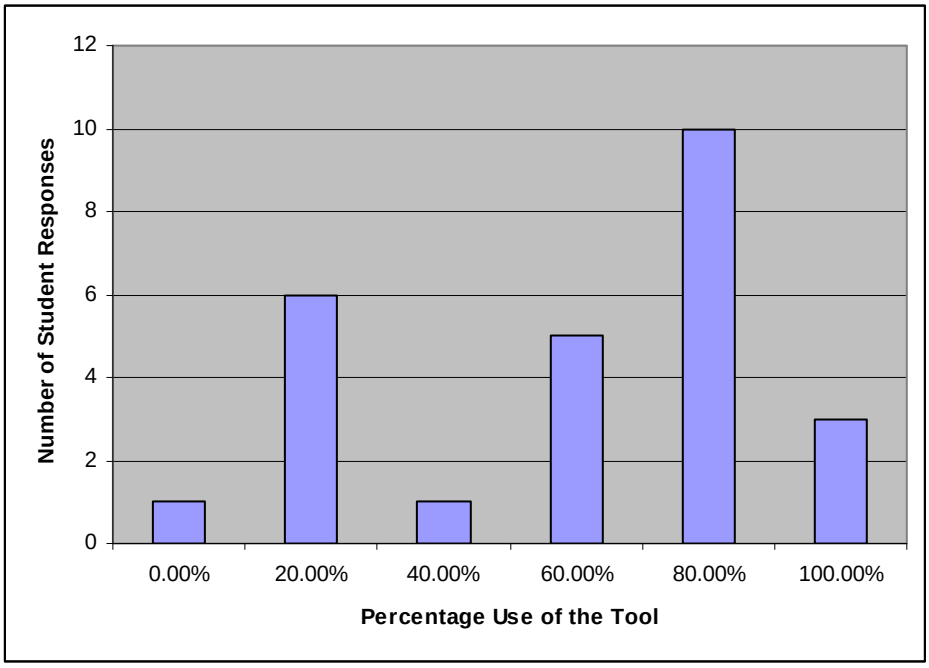


Figure 5.1. Students' percent usage of the process.

Eighteen students -- slightly less than 70% of the group -- indicated that they used the process 60% or more for working out their most recent program, as opposed to 8 students, or just over 30% of the group which includes one student who indicated no use of the process at all.

5.2.2 Development Percentages

To the question, "How much do you think the Five Step Programming Process helped you develop your most recent laboratory program?", fifteen students, or about 57.7% of them stated that the process provided 60% or more of the help with developing their most recent program. In spite of only one student indicating no use of the process, five students indicated that the process did not help them with their program development, and three students each indicated that it provided 20% or 40% of the development help. Figure 5.2 shows the distribution of the students' perceived developmental help from the process.

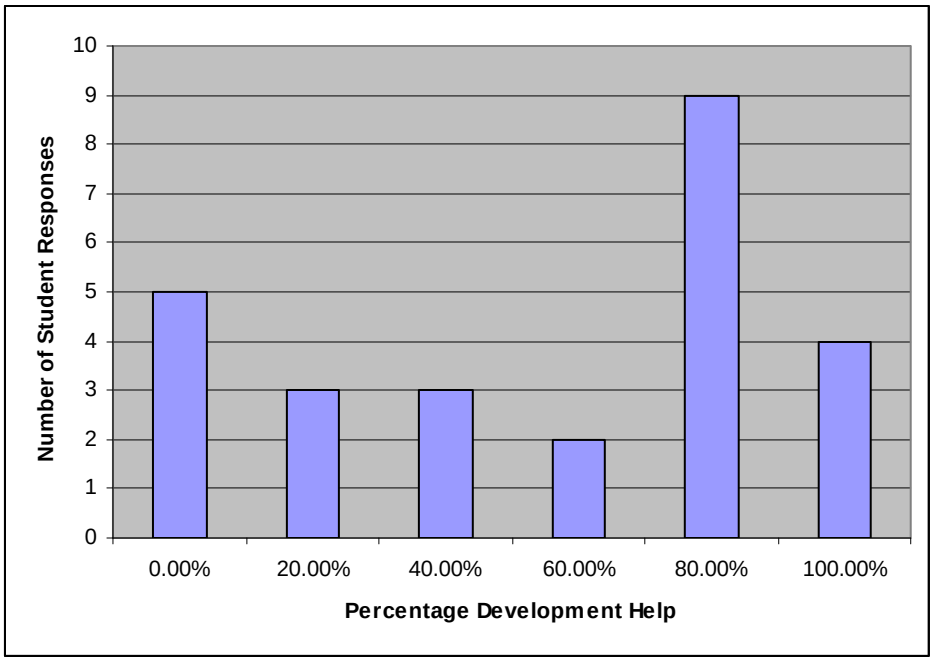


Figure 5.2. Students' perceptions of development help from the process.

5.2.3 Time Taken Percentages

This question did not ask students to estimate relative percentages. Instead, the question, "Do you think that using the Five Step Programming Process allowed you to complete your most recent laboratory program in more, less, or about the same amount of time?" provided relative or comparative answers, such as "it helped me complete the program in a little less time", or " it caused me to take a lot more time to complete my program". The range of responses is shown in Figure 5.3.

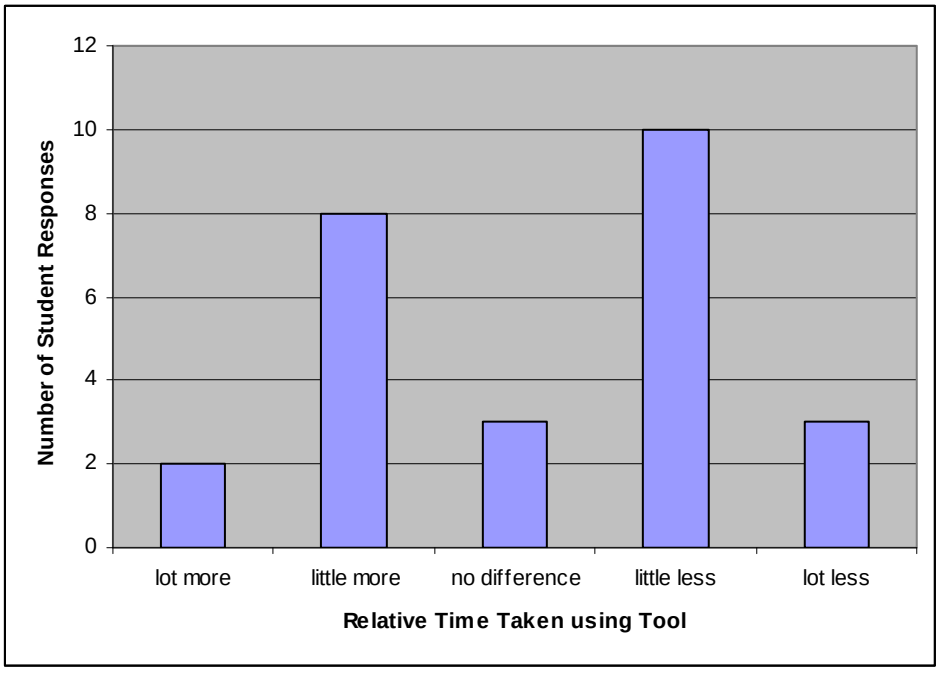


Figure 5.3. Students' perceptions of time taken to complete their laboratories

While Figure 5.3 shows bimodality, it is almost centered on the middle showing that 8 students or 30.8% of them felt that it took a little more time, and 10 or 38.5% of them felt it took a little less time. The perceived difference between taking a lot less time

and taking a lot more time only varied by one student, and the students reporting no difference in the time taken was also within one student.

5.2.4 Learning Help Percentages

The question, "How much do you think that using the Five Step Programming Process helped or hindered your learning in this programming course so far?" elicited the strongest response of all of the multiple-choice questions. Nineteen of the 26 students, or slightly more than 73% of the students responded with "it helped me learn about programming a little", or "... a lot". Figure 5.4 shows that the remaining students diminished from " it didn't make a difference with my learning to program" down to " it was a hindrance to my learning to program".

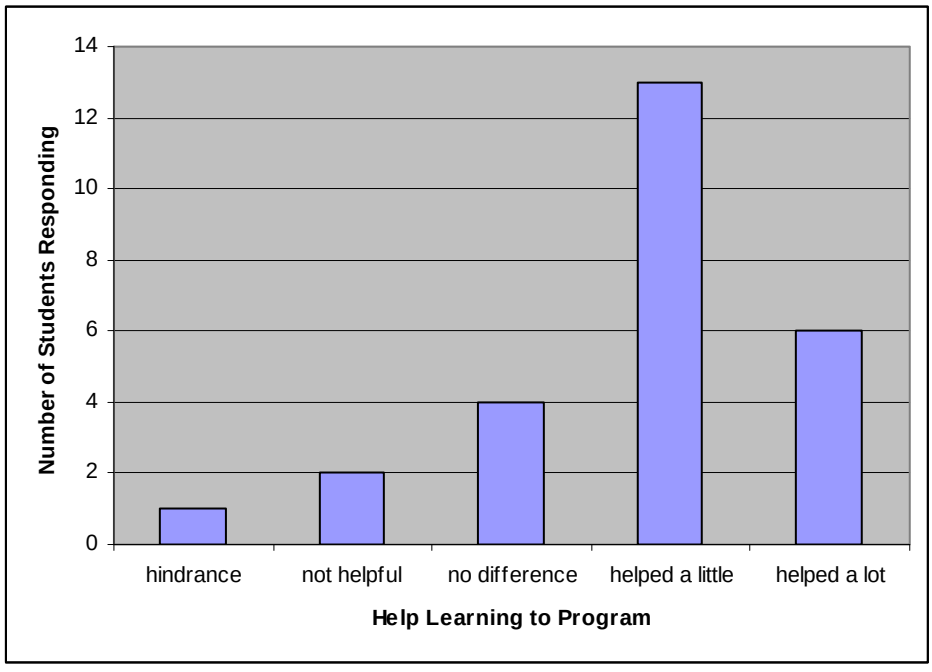


Figure 5.4. Students' perceptions of learning help provided by the process

5.2.5 Relationships Between Quantitative Data

As specified in the previous chapter, each of the quantitative items were tested for correlations to the students' actual usage of the process. Some of the correlations were promising; however the student numbers were not large enough to demonstrate significance (i.e., that the correlation was not a random result in a normal population). Since non-significant correlations are not considered to be verifiable, the relationships will be reported here, but caution and consideration must be applied to the results since none were found to be significant.

Using the Pearson Correlation Coefficient, the relationship between usage of the process and how much it supported development was .588 with a significance (two-tailed) $\leq .220$, and while the significance could not validate the results, it would appear that a greater number of students might strengthen this result. The relationship between usage and the time taken was not strong and the significance was also not very close (.450 with a two-tailed significance $\leq .447$); this result was not very promising. The strongest relationship was the correlation between usage and the learning help that the process might provide. A correlation coefficient of .712 with a two-tailed significance of $\leq .178$ brought this relationship closest to potential significance. With a larger sample, a relationship between these two variables might be found.

5.3 Student Written Responses

Students were offered the opportunity to provide their opinions on difficulties they might have had with the process, use of the process in other areas, suggested modifications to the process, and finally, they were offered an open-ended question to contribute any further ideas or opinions they might have related to learning with and using the programming process. The questions were kept simple and the students were asked to "explain briefly" their ideas and/or opinions. The student responses were grouped by general category of the response and are reported in this section. Note that the student response quotations were copied directly from the data collection results, and no grammatical or spelling changes were made.

5.3.1 Difficulties

The question, "Did you have any difficulties when using the Five Step Programming Process for developing your most recent program?" was asked of the students with a request to provide a brief explanation. Of 23 responses, 9 of them indicated that there were no significant difficulties. Two further students indicated that they did not like how it worked at first, but began to appreciate it later. For example, one student stated, "Some times it seems like it generated more busy work when the programs were simple but when they became more complex it really helped".

Three students found it hard to complete the process, with statements such as, "...because i would forget areas of the program that needed to be implented until the final step so i would have to back and add code in.", or "When you're adding new information

to an assignment based on the 5 step programming process design assignment, or if the design assignment was incomplete, it's discouraging and more time consuming having to edit additional information related to the programming assignment".

Three students felt that writing code before drawing out the design would be the more appropriate way to develop a program. One example statement was, " I always want to start writing out the code while commenting it out because I feel like I'll forget my idea later." Another student stated, "The 5 step process is not the best way to write the program. Writing down what one plans to do to complete the entire program before writing code is a bad idea.".

Five students found the process difficult to do or to understand. Representative responses were, "It's a good concept, but it's difficult to actually do and it doesn't always help.", and "sometimes your brain would naturally take you off course and it would be hard to get your mind back into the way the five step process wanted you to do it.". One interesting response showed what the student perceived to be a difference in programming method, as opposed to not being able to abstract a solution: "It is more difficult for me to think from a top down manner than from a tool use method. I would rather build the smaller components before and then utilize the working tools then try to do an overall and guess at what I will need. Finally, one student simply stated, "step 5 is hard".

The only other response to this question was a commentary on how the programming process might be taught because the student indicated that s/he did not "understand the process at first". One of the students who indicated no difficulties also made a recommendation on how the process might be taught more clearly.

5.3.2 Application of the Process to Non-Programming Areas

One of the goals of teaching problem solving targets the goal of transfer to other problem-solving conditions that are not directly related to the original context. The question, "Do you think using the Five Step Programming Process might help with other courses, such as math, science, etc?" was designed to find out if students might see the value of using this standardized process in other problem-solving venues.

Only two of the 26 students responded positively and directly to the question. Both responses demonstrated recognition and application of the process: "I think the methodology behind it, taking a big problem and splitting that problem into little parts, is uesable whenever you want to solve any problem." and "Yes, I believe it's applicable to other courses and I have used it in other courses for projects and essays.". The positive responses were separated in the research from a second group who believed it might be useful elsewhere, but either had not tried it, or may have not thought about it prior to answering the question.

There were 15 students who considered that it was possible that the process -- or its parallel practice -- might be used elsewhere. Example student responses are: "Maybe writing English essays though I haven't tried it.", "The program could really be used for any other class. It is basically saying what you should do in an assignment then actually doing it.", "The process itself seems to be related to only computer science; however, the logic behind it definitely carries over into courses of other fields such as mathematics and physics. The idea of finding the main goal of a problem and breaking it down into smaller problems is essential to both this course and others.", and "Possibly for really complex

problems for calculus. I've actually never really thought about it." Again, these responses were not accepted as directly positive answers to the questions, but were accepted as students who appeared to be considering the possibility that the process might be transferrable.

There was only one unqualified "no" in the remaining group of responses. Like the previous group, the remaining students stated "no" or "I don't think it will", and then expanded on their response with their own considerations. Example student responses are: "I do not think the process will be helpful with other courses in its specific structure. I think it does help to teach how to break a problem down into smaller chunks.", "not really in math, but it could be helpful to developing an idea in a science.", "No i dont think it would be useful. Since most other courses do not solve problems in a manner that can be modularized.", and "no, I won't be taking other courses like that".

Two more directly negative responses still seemed to be considering the use of the process, even if they did not think it would work elsewhere: "no For math, one would not write down pages of commentary on how they thought they were going to solve the problem before attempting the problem.", and " No, it is more or less a step made for planning, which I could do just as well without this five-step process. In fact, I would find it a large bother to translate its application to something other than coding, given my other subjects are more about learning specific equivalents of functions rather than a large algorithm."

5.3.3 Condensing Steps Together

To find out what the students think about the individual steps, and the overall stepwise process as a whole, one question was asked about condensing two or more steps into one, and the other was asked about expanding any of the steps. The data from these are treated separately, and the condensing down question is presented here. Eleven of 24 students indicated that the steps were okay as they were. Example responses were: "I like keeping the steps individually, as taken one step at a time it develops very well.", "It's great the way it is because of the way each step builds on the previous one." These responses address the goal of stepwise progress toward the solution, but one student addressed the motivational and to some extent the psychological considerations: "Having 5 steps is a good way to go, mainly because it helps the student focus on the problems at hand. Stepping through each large problem, and solving smaller problems to accomplish it relieves a lot of stress. Also, the feeling of accomplishment after each step acts as a great motivator to continue through the rest of the program."

The question for this item was, "Are there parts of the five steps you think should be condensed down? For example, because step 1 is so brief, do you think it would help to combine steps 1 and 2?" Because the example given is actually a consideration, this question is somewhat leading. This was done purposely to see if students would argue for or against condensing questions 1 and 2 into one. Some students concurred, but apparently for reasons of managing the larger process: "I think consolidating 1 and 2 would be helpful because if anyone else is like me, they don't remember the specifications of all five steps. I pretty much know what step 4 is supposed to look like

from the design assignments.", and "Combining steps 1 and 2 would be helpful in that the steps would all be very distinct; it would be easier to follow if you knew exactly what step you're on. In my mind the line between step 1 and 2 is not clear."

As was found in a previous question, there were students who felt that working out the program before coding was not important: "In my personal execution of the process, I would almost always start at step 3/4. The reason is because, in my opinion, the first 2 steps outline things that I don't really feel are too confusing or complex to need commenting. (i.e. basic understood functions, obvious coding needed in main()). However, steps 3 and 4 are great in helping with the complexity of the code that can arise inside functions. (i.e. loop iteration, array augmentation, etc.)", and "I think steps 1, 2, and 3 can be condensed down into one step. In the beginning, it helped a lot to be able to do 5 steps but as the semester carries on, step one and two become vestigial steps."

One student suggested removing the function specifications part of one step: "Step 3: function specifications. I'm usually only guessing what my functions are or what they can do. To base my program in step 4 around these functions can be bad. I'm a think as I go person, planning things out too far ahead usually results in bad work. I would cut step 3 out, and just go from step 2 to step 4." Another student suggested condensing steps 4 and 5 due to problems with later program modification: "The only complaint that I have about the 5 step process is step 4 and 5 should be condensed because often times I find that I need to adjust or modify my main program code when I am writing my functions which can often render my step 4 work useless."

The last group of students did not see the value of a stepwise process: "I never do this step on at a time. I think it would be ridiculous to do them one at a time. So yes I

think they could be condensed down.", "outline work on the functions and what they are doing place the funcitons where they need to go comment the functions finish everything", and "I like to break down problems as I see fit for each individual problem, it is a waste of time to have a single approach that you try to use for every problem since many problems are handled differently".

5.3.4 Expanding Steps Out

Of 24 responses, 15 of them felt that no expansion was necessary, most with little qualification, and some with rationale for keeping them as the are: "I think the expanding is not beneficial at this level of CS", " No, I think step 3 and 4 are great as they are, because they help you get prepared for coding in a fairly concise manner, and I see no reason to arbitrarily expand on the process.", and "Commenting out the stub functions in step 4 helps keep track of what the main program is supposed to do; therefore, I believe it should be left the same.".

The question for expanding out the process was, "Are there parts of the five steps that you think should be expanded upon? For example, because you write both the main program and the stub function comments in Step 4, do you think it would help to break this process into two unique steps?" This was also somewhat leading, but again this was a clear example to the students as to what was being requested. Three students did suggest breaking up step 4, with specific arguments: "There is a lot in step 4. Breaking it up would help.", "The only aspect that could be elaborated on would be step 4. I found it difficult to know what exactly i wanted a function to do and how it was going to be implemented.", and "Step four is very big, and should be break down in multiple parts".

The remaining responses to this question were not so much related to expanding steps, but are added here for completeness, and will contribute to the discussion in the next chapter: "I do think that stubbing out the functions should be step 4 and writing main should be added to step 5.", and "Step 2 should include that it is a form of step 1 applied to segments and then reapplied until the function is split into small enough pieces." One student apparently continued supporting the idea that all the coding should happen as it is approached: "It would help to actually write the functions instead of just stubbing them. Sometimes, it would help wirting the main program."

5.3.5 Other Comments

In order to provide students an opening to provide any other ideas, suggestions, issues, and so on, an open-ended question, "If you have any other opinions or thoughts on your use of the Five Step Programming Process, please share them here.", was provided at the end of the survey. There were a variety of responses starting with four students who indicated that they had no issues to discuss. Four students stated that they liked the process, although two of them offered suggestions about how to improve the delivery of it: "It's a good process, but it could use a better description online. The steps listed above in this survey is a nice summary. Put that online.", and "This process is a very powerful tool, and is helpful in writing efficient code. The only suggestion I have is to introduce the process earlier in the course, saving students much frustration toward the class, and even the instructor :)".

Two students stated that it worked for them, but with some qualifications: "Originally I hated having to use the programming process as I felt it made me spend

extra time outlining when I didn't need to. This was because I hadn't really encountered anything in class that was really of any conceptual difficulty. When we started getting into loop, array, and file streaming design, the use of the process really became more important in my programming. Once I embraced actually taking the time to outline everything it really helped me conceptualize code structure and potential problems a whole lot easier. I really feel like a much stronger coder now, and I am very glad that I came around to using the programming process.", and "The process is tediuous, and certainly not something I enjoy doing, however it allowed my programs to come out looking relatively clean and usually I understood more of my program coming out than going in."

Four students did not see the value of the process. One student, who felt that many of his classmates agreed with him stated, "…Many students don't use their design assignments at all when completing their programming assignments. I use mine, but I don't follow the process to complete the design assignment, I just make it follow the format in the end.". Another student believed that other methods should be offered for use: "People's minds work differently. All students should not be required to follow the same formula. If the students understands what is needed to be done to achieve a goal (writing a program), they should not be forced to do it a certain way. The students should be shown different methods and be allowed to chose which method works best. CS 135 would be a better class if the students were only graded on the programming assignment and not the design assignment."

Another student did not see the value of integrating the comments into the program: "For a beginner to programming, the five step process is solid. It makes

readable, easy to follow code for you and other users. Later on, I would be more concerned with finishing a programming assignment rather than completing a design assignment that may not accomplish my original goals, and therefore be scrapped. I'm essentially writing my thoughts down in english with the 5 step programming process. If I'm doing that, why can't I just write down my thoughts in english outside of the program?" The last of the students who did not like the process stated, "I found actually coding a skeleton of the main program while commenting what I was doing was most effective. From there, I would add on flags, additional features, and the like. Even while doing a design assignment and no actually coding, I still confused myself with all the parts going on. I think having a partial program running helped me see what I did and did not have, which allowed me to handle the next task without thinking about how it interacted with every other task. I'll point out that when just planning, I was not sure if all the parts would mesh correctly which led to a lot of mental stress over handling the seperate parts appropriately. Step 1 should take care of this, but it did not. Finally, one program I wrote out the entire design assignment, then could not follow it when I had to code it. Instead, I simultaneously coded and commented it. When I compared the comments, they were almost exactly the same."

Two more students argued that the process should be optional, one simply making the statement, and the other stating, "I think it should be optional. I don't see the need for this tool, in fact I find my self witting the code as I do it anyway, going back commenting on it then deleting the code I wrote. this seems very counter productive. I guess it might help some people but I really had no use for it." Three students offered further suggestions for how to introduce or teach the process, and three other students

commented on how hard the lab was, how s/he was graded, and in one case, the feeling that s/he was not getting it right.

5.3.6 Chapter Conclusion

This chapter reported on both the quantitative data, and examples of the written data, that were collected from the students. As was noted, the number of students was too small to show significance in the quantitative data, but as a pilot study, both the quantitative data and the student comments provided information and insight from the students as to their perceptions of, and interaction with, the programming process tool. In addition, all of this data shows promise if a comparable study could be re-implemented with a larger student population. In the next chapter, this data will be evaluated with consideration for how the students were thinking, and how the process might be adapted to become more effective for their use.

Chapter 6
Conclusions and Future Work

This research project represents a first step pilot assessment of the use of the Five Step Programming Process. While it was primarily meant to get a first layer look at what students thought about the process, some interesting results were found. There were students who considered the process helpful, and there were students who did not understand, or outright rejected, the use of the process. The data from both groups is important to future work in this area. This chapter will present discussion and tentative conclusions on the information from the research, and then -- with the understanding that this pilot activity was specifically designed to be continued -- the chapter will discuss how the knowledge found in this study can support further research. Finally, a brief discussion will be provided related to the academic areas, journals, and conferences to which the results of this research may be directed.

6.1 Concluding Remarks

A large percentage of students stated that they used the process on the most recent programming project. This is promising, but could be a result of being required to provide part of their homework in the step 4 and later the step 5 format. However, some students noted that they simply organized the homework to fit the format without using

the process, and one student reported that s/he did not use the process at all. Thus there is evidence that the students reporting usage really did take advantage of the process.

The assistance with program development also showed some promise, but showed quite a bit of variability among the students. It was somewhat surprising that the time taken by the students did not show a better response (i.e., more evidence of less time required for a program). In the previous semester, the most often anecdotally reported benefit of using the process was the time savings. At least two of the potential variables here are: 1) classes of students do vary from semester to semester; as an example, the previous semester contained a group of Honors students; and 2) students during this semester -- again anecdotally -- seemed to have more trouble with the length of time the projects required than previous semesters.

The relationships found between usage of the process and development assistance, and usage of the process and learning help were both promising. As stated in the previous chapter, these values were not found to be significant and therefore cannot be relied upon for a firm conclusion. Nevertheless, the correlations and the levels of significance found hint that for a larger scale research study, the statistical evidence might be strengthened for both. If this could be verified, there would be evidence of the process' support of the double goal of both improved programming and improved learning.

Beyond the quantitative data acquired, the students' written responses were enlightening, and as mentioned, both the positive and the negative responses provided important feedback. Almost 40% of the students had no difficulty with the process, but the students who did mention problems identified issues that the process should have

mitigated had they used it appropriately. Students who stated that they needed to write code first so they would not forget and students who stated that they would get off course were not experiencing the value of the advanced and focused thinking that this kind of process should have provided them.

The question related to use of the process outside of programming was a gentle probe to seek student thought on how organized and/or systematic planning might work elsewhere, and as an assessment tool, it was effective. Sixty percent of the students made statements about how the process might work elsewhere. These responses demonstrated hypothetical thinking about where else an advanced planning process might work. In fact, most of the negative responses still hypothesized how the process might or might not work, which is evidence that they were understanding the overlying structure of this kind of process.

As far as modification of the process, no strong feedback was provided for making significant changes. Even with the slightly leading example of condensing steps 1 and 2 of the process, the students who agreed did so because they were not feeling completely clear on how the steps worked uniquely within the whole process. Virtually all of the remaining students who argued for reducing steps did so from the perspective that they did not see the value of a complete plan before beginning the coding process. This is of significant interest and will be discussed next.

Students arguing for expansion of the process were fewer. Sixty percent of these students stated that there was little or no need for an expansion of the steps. However, some students did support the somewhat leading example of condensing steps 4 and 5 although again, the arguments tended to be toward getting into the coding sooner. The

contraction and expansion questions both seemed to identify the problem that the programming process is focused on resolving. Many of the student responses in this data demonstrate the apparent student "need" to start coding without serious consideration for an organized planning process.

The responses were almost dichotomous between the students who acquired an understanding of the value of the process, "Once I embraced actually taking the time to outline everything it really helped me conceptualize code structure and potential problems a whole lot easier.", and the students who did not, "I found actually coding a skeleton of the main program while commenting what I was doing was most effective." This second response represents students who do not understand the problems with designing or managing a non-trivial program, and will continue to be the target at which teaching a development procedure will be aimed.

Probably the most telling response from a student unfamiliar with designing non-trivial programs was the statement, "Writing down what one plans to do to complete the entire program before writing code is a bad idea.". Whether the student thought through this statement before typing it or not, it is the best evidence that students -- and not just programming students -- must be taught the value of advance planning and/or design. It might be conjectured that this is a capable or possibly advanced student who has been able to create the mental model required of the moderately non-trivial programming assignments assigned in the CS1 course, but who has not been challenged by a program at a scale that would demonstrate the value of advanced design.

In any event, the results of this pilot study have been helpful. The primary goal of the research was to get feedback from the students using, and potentially learning with,

the Five Step Programming Process. The feedback seems to be both positive and supportive; it appears that students are gaining value from use of the process. In addition, no significant evidence has been provided to support major changes to the process. There was some feedback requesting better communication (i.e., teaching) of the process, and the somewhat unexpected outcome was the fairly strong evidence that this kind of process is needed.

6.2 Future Improvements and Research

Since this was a pilot study and was purposely limited to building a background or framework for future studies, there are a number of modifications to teaching and identified areas for further research generated from this research project. Some of these are provided in this section.

6.2.1 Changes to the Educational Process

The first significant action that will be driven by this research is the development of a more complete, but somewhat simpler training system for use of the process. The process was presented briefly in class, and supported in the laboratories and with a web site. Responses to this research show that the students need both a more concise overview of the process as well as more examples of its use.

The second action made in direct response to some of the feedback will be to start teaching the process earlier in the course albeit in a simplified way that will not involve the use of concepts to which they will not have been exposed to at first (e.g., functions). Using the process in the classroom from the earliest part of the semester will make the

students comfortable with it at an earlier stage, and more prepared for it when the programs become complex enough to require modular breakdown.

The third, and probably the most needed action, is problematic. As with virtually any other learners, beginning CS1 students must start out with trivial or non-complex programs in order to gain the experience that will become mental and long-term memory constructs later. While this "training wheels" approach is necessary and seemingly appropriate, it may lead to the result found in this research that some students believe they do not need a design process for their programs. The programs assigned in this particular CS1 course tend to be moderately non-trivial as soon as the students have learned about some of the basic tools of programming.

However, this appears to be where the dichotomy mentioned previously opens up. From frequent student feedback, both anecdotally and formally provided in course evaluations, many of the students say that the programming assignments take too much time outside of class. On the other hand, this research shows that there are still students who feel -- somewhat strongly in some cases -- that they can develop these programs without a design process. As mentioned in Chapter 4, this has led some educators to want to teach the "programming in the large" process, but as the research showed, this strategy has not resulted in significant improvements, and the "programming in the large" process loses the value of trying to solve a problem from start to finish with a program. This would seem to make the CS1 course more an industrial programming course, and less a problem-solving course.

As also mentioned in Chapter 4, the problems with teaching this course are not trivial, but identifying these problems allows educators to focus on, analyze, and attempt

to resolve them. New strategies need to be found to address this part of the CS1 educational endeavor, and this research provides some support for how to develop these strategies.

6.2.2 Future Research

While the focus is on problem solving and programming, there are several supporting areas that can be extended with further research. The cognitive load component [6], and the related working memory capacity issues [21] that are pertinent to this project and research are well supported, and continue to be actively pursued. Conversely, components such as Gilmore's [28] study of program organization, Hartley's [33] study of white space, and Payne, Sime, and Green's [43] study of perceptual characteristics of a program need further consideration. It may be possible to identify conditions of understanding as a result of organization and perception with individuals other than student programmers, and this would likely be more generalizable to educational practices beyond introductory programming.

As mentioned in the previous section, it will be critical to get students to use and trust the programming process, but it will be more critical that the organization and structure of the process helps them stay on track with a reasonable planning or designing process. Research on motivational aspects of the process, or motivational strategies for using the process could be helpful to finding greater student success. It may be productive to pursue focal quantities in programming since they are apparently a natural student, or possibly human, attribute as discussed by Davies [22], Rist [50], and Robins, Rountree, and Rountree [52].

In addition, a next important step for this research would be to conduct an experimental comparison between students using this process and students using another formalized process, or none at all. This may be conducted between colleges and universities in the local area since they have the same course structures, but it can be difficult due to differences in teaching style and strategy. It is nevertheless worth the efforts that might prove necessary.

Finally, a longer term study will be conducted on students who have been exposed to this development strategy by assessing third- and/or fourth- year students with questions on their possible use of the process. Given appropriate numbers of students, comparison studies between students who have learned this strategy and students who have not should be conducted. Another interesting study that would be logistically challenging, but potentially rewarding would be to study the non-Computer Science students such as the Electrical Engineering, Physics, or Mathematics students who were required to take the CS1 class but very likely did not continue with a significant amount of programming.

6.2.3 Dissemination of the Research

There is a significant amount of discussion related to integrating CS1 into high school and/or K-12 education [34] at this time, although as discussed in Chapter 1, there is not much research support to back this action up right now. In addition, there continues to be much discussion related to problem solving in technical and other fields [35]. This provides a number of opportunities to contribute research related to CS1 teaching to the scientific and education communities.

For example, the Association of Computing Machines' (ACM) Special Interest Group: Computer Science Education (SIGCSE) publishes four large bulletins per year focused on this topic, as does the ACM *Transactions on Computing Education*. In addition, the Consortium for Computing Sciences in Colleges (CCSC) publishes four Journals and conducts regionalized conferences every year as well. Other venues that directly address technology education are the *IEEE Transactions for Education* and the related annual Frontiers of Education (FIE) conference; and the American Society of Engineering Education (ASEE) which also supports the FIE conference, sponsors both the *Prism* magazine that includes a teaching component, and the *Journal of Engineering Education*. These Engineering publications are not focused only on Computing Science, but CS articles are among others that are published.

All of the above publications are likely forums for disseminating research such as that found in this thesis, and parts of this research will be proposed to one or more of them. This document began with a discussion of the difficulties -- and indeed the failures -- with teaching introductory Computer Science. It continued with evidence that just teaching people to program does not necessarily support higher level, advanced thinking, or even programming particularly well. However, some of the research showed that teaching the thinking process and the programming process together can provide effective support toward helping students program and solve problems. This research has found preliminary evidence that a process can help students learn effective programming skills and gain a larger-scale picture of solving non-trivial problems. This project and future research can offer contributions to the educators and researchers working in this

arena, and to the students who benefit from the knowledge that is gained through these endeavors.

Bibliography

[1] Afsharian, S., Giacomobono, M. and Inverardi, P. A framework for software project estimation based on cosmic, dsm and rework characterization. In *Proceedings of the 1st international workshop on Business impact of process improvements* (Leipzig, Germany, 2008). ACM, City, 2008.

[2] Allan, V. H. and Kolesar, M. V. Teaching Computer Science: A Problem Solving Approach that Works. *SIGCUE Outlook*, 25(1,2): 2-10, 1997.

[3] Anderson, J. R. *Rules of the Mind*. Lawrence Erlbaum Associates, Publishers, Hillsdale, NJ, 1993.

[4] Anderson, J. R. and Jeffries, R. Novice LISP Errors: Undetected Losses of Information from Working Memory. *Human-Computer Interaction*, 1(2): 107-131, 1985.

[5] Ayres, P. L. Why Goal-Free Problems Can Facilitate Learning. *Contemporary Educational Psychology*, 18(3): 376-381, 1993.

[6] Barrouillet, P., Bernardin, S., Portrat, S., Vergauwe, E. and Camos, V. Time and Cognitive Load in Working Memory. *Journal of Experimental Psychology: Learning, Memory, and Cognition*, 33(3): 570-585, 2007.

[7] Beaubouef, T. and Mason, J. Why the high attrition rate for computer science students: some thoughts and observations. *SIGCSE Bulletin*, 37(2): 103-106, 2005.

[8] Bishop-Clark, C. Cognitive Style, Personality, and Computer Programming. *Computers in Human Behavior*, 11(2): 241-260, 1995.

[9] Boehm, B. A View of 20th and 21st Century Software Engineering. In *Proceedings of the 28th International Conference on Software Engineering* (Shanghai, China, 2006). ACM, City, 2006.

[10] Bransford, J. D., Brown, A. L. and Cocking, R. R. *How People Learn: Brain, Mind, Experience, and School*. National Academy Press, Washington, D.C., 1999.

[11] Brooks, R. Categories of Programming Knowledge and Their Application. *International Journal of Man-Machine Studies*, 33(3): 241-246, 1990.

[12] Brooks, R. Towards a theory of the cognitive processes in computer programming. *International Journal of Man-Machine Studies*, 9(6): 737-751, 1977.

[13] Brooks, R. Towards a Theory of the Comprehension of Computer Programs. *International Journal of Man-Machine Studies*, 18(6): 543-554, 1983.

[14] Bruce, C., Buckingham, L., Hynd, J., McMahon, C., Roggenkamp, M. and Stoodley, I. Ways of Experiencing the Act of Learning to Program: A Phenomenographic Study of Introductory Programming Students at University. *The Journal of Information Technology Education*, 3(1): 143-160, 2004.

[15] Bunch, J. M. An Approach to Reducing Cognitive Load in the Teaching of Introductory Database Concepts. *Journal of Information Systems Education*, 20(3): 269-275, 2009.

[16] Carlisle, M. C. Raptor: A Visual Programming Environment for Teaching Object-Oriented Programming. *Journal of Computing Science in Colleges*, 24(4): 275-281, 2009.

[17] Carlisle, M. C., Wilson, T. A., Humphries, J. W. and Hadfield, S. M. RAPTOR: A Visual Programming Environment for Teaching Algorithmic Problem Solving. *ACM SIGCSE Bulletin*, 37(1): 176-180, 2005.

[18] Catrambone, R. The Subgoal Learning Model: Creating Better Examples So That Students Can Solve Novel Problems. *Journal of Experimental Psychology: General*, 127(4): 355-376, 1998.

[19] Chi, M. T. H., Bassock, M., Lewis, M. W., Reimann, P. and Glaser, R. Self Explanations: How Students Study and Use Examples in Learning to Solve Problems. *Cognitive Science*, 13(2): 145-182, 1989.

[20] Clear, T. "Programming in the Large" and the Need for Professional Discrimination. *SIGCSE Bulletin*, 33(4): 9-10, 2001.

[21] Cowan, N. The Magical Number 4 in Short-Term Memory: A Reconsideration of Storage Capacity. *Behavioral and Brain Sciences*, 24(1): 87-114, 2001.

[22] Davies, S. P. Models and Theories of Programming Strategy. *International Journal of Man-Machine Studies*, 39(2): 237-267, 1993.

[23] Donovan, M. S. and Bransford, J. D. *How Students Learn: Mathematics in the Classroom*. The National Academies Press, City, 2004.

[24] Eisenstadt, M. and Brayshaw, M. (1989). An Integrated Textbook, Video, and Software Environment. In J. C. Spohrer and E. Soloway (Eds.), *Studying the Novice Programmer* (pp. 447-466). Hillsdale, NJ: Lawrence Erlbaum Associates, Publishers.

[25] Gerjets, P., Scheiter, K. and Catrambone, R. Designing Instructional Examples to Reduce Intrinsic Cognitive Load: Molar versus Modular Presentation of Solution Procedures. *Instructional Science*, 32(1-2): 33-58, 2004.

[26] Giangrande, E., Jr CS1 Programming Language Options. *Journal of Computing Sciences in Colleges*, 22(3): 153-160, 2007.

[27] Gilmore, D. J. Comprehension and recall of miniature programs. *International Journal of Man-Machine Studies*, 21(1): 31-48, 1984.

[28] Gilmore, D. J. *Structural Visibility and Program Comprehension*. Cambridge University Press, City, 1986.

[29] Goldensen, D. Why Teach Computer Programming? Some Evidence about Generalization and Transfer. In *Proceedings of the Annual National Educational Computing Conference* (Minneapolis, MN, 1996), City, 1996.

[30] Green, T. R. G. Conditional program statements and their comprehensibility to professional programmers. *Journal of Occupational Psychology*, 50(2): 93-109, 1977.

[31] Green, T. R. G. (1990). Programming Languages as Information Structures. In J.-M. Hoc, T. R. G. Green, R. Samurcay and D. J. Gilmore (Eds.), *Psychology of Programming* (pp. 117-138). San Diego, CA: Academic Press Inc.

[32] Green, T. R. G., Bellamy, R. K. E. and Parker, M. (1987). Parsing and Gnisrap: a model of device use *Empirical studies of programmers: second workshop* (pp. 132-146): Ablex Publishing Corp.

[33] Hartley, J. Spatial Cues in Text: Some Comments on the Paper by Frase & Schwartz. *Visible Language*, 14(1): 62-79, 1980.

[34] Hazzan, O., Gal-Ezer, J. and Blum, L. A model for high school computer science education: the four key elements that make it! *ACM SIGCSE Bulletin*, 40(1): 281-285, 2008.

[35] Kiesmuller, U. Diagnosing Learners' Problem-Solving Strategies Using Learning Environments with Algorithmic Problems in Secondary Education. *ACM Transactions on Computing Education*, 9(3): 17:11-17:26, 2009.

[36] Kirkwood, M. Infusing higher-order thinking and learning to learn into content instruction: a case study of secondary computing studies in Scotland. *Journal of Curriculum Studies*, 32(4): 509-535, 2000.

[37] Lyu, M. R. Software Reliability Engineering: A Roadmap. In *Proceedings of the 2007 Future of Software Engineering* (2007). IEEE Computer Society, City, 2007.

[38] Mayer, R. E., Dyck, J. L. and Vilberg, W. (1989). Learning to Program and Learning to Think: What's the Connection? In E. Soloway and J. C. Spohrer (Eds.), *Studying the Novice Programmer* (pp. 113-124). Hillsdale, NJ: Lawrence Erlbaum Associates.

[39] McCracken, M., Almstrum, V., Diaz, D., Guzdial, M., Hagan, D., Kolikant, Y. B.-D., Laxer, C., Thomas, L., Utting, I. and Wilusz, T. A Multi-National, Multi-Institutional Study of Assessment of Programming Skills of First-Year CS Students. *ACM SIGCSE Bulletin*, 33(4): 125-180, 2001.

[40] Merrill, M. D. First Principles of Instruction. *Educational Technology Research & Development*, 50(3): 43-59, 2002.

[41] Ormerod, T. (1990). Human Cognition and Programming. In J.-M. Hoc, T. R. G. Green, R. Samurcay and D. J. Gilmore (Eds.), *Psychology of Programming* (pp. 63-82). San Diego, CA: Academic Press Inc.

[42] Paas, F., Renkl, A. and Sweller, J. Cognitive Load Theory and Instructional Design: Recent Developments. *Educational Psychologist*, 38(1): 1-4, 2003.

[43] Payne, S. J., Sime, M. E. and Green, T. R. G. Perceptual Structure Cueing in a Simple Command Language. *INternational Journal of Man-Machine Studies*, 21(1): 19-29, 1984.

[44] Pea, R. D. and Kurland, D. M. On the Cognitive Effects of Learning Computer Programming. *New Ideas in Psychology*, 2(2): 137-168, 1984.

[45] Pears, A., Seidman, S., Malmi, L., Mannila, L., Adams, E. S., Bennedsen, J., Devlin, M. and Paterson, J. A Survey of Literature on the Teaching of Introductory Programming. *ACM SIGCSE Bulletin*, 39(4): 204-223, 2007.

[46] Pennington, N. (1987). Comprehension Strategies in Programming. In G. M. Olson, S. Sheppard and E. Soloway (Eds.), *Empirical Studies of Programmers: Second Workshop* (pp. 100-113). Norwood, NJ: Ablex Publishing Company.

[47] Perkins, D. N., Hancock, C., Hobbs, R., Martin, F. and Simmons, R. (1989). Conditions of Learning in Novice Programmers. In E. Soloway and J. C. Spohrer (Eds.), *Studying the Novice Programmer* (pp. 261-280). Hillsdale, NJ: Lawrence Erlbaum Associates.

[48] Perkins, D. N. and Martin, F. (1986). Fragile Knowledge and Neglected Strategies in Novice Programmers. In E. Soloway and S. Iyengar (Eds.), *Empirical Studies of Programmers: Papers presented at the First Workshop on Empirical Studies of Programmers June 5-6, Washington, DC* (pp. 213-229). Norwood, NJ: Ablex Publishing Company.

[49] Renkl, A., Atkinson, R. K., Maier, U. H. and Staley, R. From Example Study to Problem Solving: Smooth Transitions Help Learning. *The Journal of Experimental Education*, 70(4): 293-315, 2002.

[50] Rist, R. S. Program Structure and Design. *Cognitive Science*, 19(4): 507-562, 1995.

[51] Rist, R. S. Schema Creation in Programming. *Cognitive Science*, 13(3): 389-414, 1989.

[52] Robins, A., Rountree, J. and Rountree, N. Learning and Teaching Programming: A Review and Discussion. *Computer Science Education*, 13(2): 137-172, 2003.

[53] Rogalski, J. and Samurcay, R. (1990). Acquisition of programming Knowledge and Skills. In J.-M. Hoc, T. R. G. Green, R. Samurcay and D. J. Gilmore (Eds.), *Psychology of Programming* (pp. 157-174). San Diego, CA: Academic Press Inc.

[54] Sattar, A. and Lorenzen, T. Teach Alice Programming to Non-Majors. *SIGSCE Bulletin*, 41(2): 118-121, 2009.

[55] Simon, B., Hanks, B., McCauley, R., Morrison, B., Murphy, L. and Zander, C. *For Me, Programming is ...* ACM, City, 2009.

[56] Sleeman, D. The challenges of teaching computer programming. *Communications of the ACM*, 29(9): 840-841, 1986.

[57] Soloway, E. Learning to Program = Learning to Construct Mechanisms and Explanations. *Communications of the ACM*, 29(9): 850-858, 1986.

[58] Spohrer, J. C. and Soloway, E. (1989). Novice Mistakes: Are the Folk Wisdoms Correct? In E. Soloway and J. C. Spohrer (Eds.), *Studying the Novice Programmer* (pp. 401-416). Hillsdale, NJ: Lawrence Erlbaum Associates, Publishers.

[59] Sweller, J., van Merrienboer, J. J. G. and Paas, F. G. W. C. Cognitive Architecture and Instructional Design. *Educational Psychology Review*, 10(3): 251-296, 1998.

[60] Tew, A. E., Fowler, C. and Guzdial, M. Tracking an innovation in introductory CS education from a research university to a two-year college. *SIGCSE Bulletin*, 37(1): 416-420, 2005.

[61] Thune, M. and Eckerdal, A. Variation theory applied to students' conceptions of computer programming. *European Journal of Engineering Education*, 34(4): 339-347, 2009.

[62] Van Merrienboer, J. J. G. Instructional Strategies for Teaching Computer Programming: Interactions with the Cognitive Style Reflection-Impulsivity. *Journal of Research on Computing in Education*, 23(1): 45-54, 1990.

[63] Van Merrienboer, J. J. G., Kirschner, P. A. and Kester, L. Taking the Load Off a Learner's Mind: Instructional Design for Complex Learning. *Educational Psychologist*, 38(1): 5-13, 2003.

[64] Winslow, L. E. Programming Pedagogy - A Psychological Overview. *ACM SIGCSE Bulletin*, 28(3): 17-22, 1996.

[65] Yuen, A. H. K. Teaching Computer Programming: A Connectionist View of Pedagogical Change. *Australian Journal of Education*, 44(3): 239-253, 2000.

Appendix A - Survey Materials

1. IRB Consent Request

UNIVERSITY OF NEVADA, RENO
SOCIAL BEHAVIORAL INSTITUTIONAL REVIEW BOARD
CONSENT TO PARTICIPATE IN A RESEARCH STUDY

TITLE OF STUDY: The Five Step Programming Process
INVESTIGATOR(S): Frederick Harris, 775.784.6571, Michael Leverington, 775.784.1414
PROTOCOL #: SB09/10-122

PURPOSE/PARTICIPANTS

If you are at least an 18 year old adult student in this semester's CS 135 course, you are being asked to participate in a pilot research study to evaluate the helpfulness and usability of a tool you are presently using in your CS 135 course. At least 30 students are expected to respond to this survey.

PROCEDURES

If you agree to participate in this research study, data from the assessment you are about to take will be used in a Master's Degree thesis research project. If you do not wish to participate in the study, you may leave this web survey at any time.

Your actions will be the following:

1. You will read this consent information and then, at your discretion, select to participate in the survey.

2. You will then answer a small number of survey questions on subsequent pages. There are four multiple-choice questions, four brief explanation questions, and one open-ended opinion question.

3. You may choose to withdraw from the research at any time by simply closing the WebCampus survey.

4. About a week after you have taken the survey, you will be able to pick up your food gift card from the Computer Science & Engineering office at SEM 242.

BENEFITS, DISCOMFORTS, INCONVENIENCES, AND/OR RISKS

1. There may be no direct benefits to you as a participant in this study but we wish to acquire your feedback so that we can improve the Five Step Programming Process learning tool.

2. As this is an opinion survey, there is very little likelihood of significant discomfort or risk, and any inconvenience would only be a result of the 10 to 20 minutes that the survey may require. None of the questions should offer any discomfort.

CONFIDENTIALITY

1. Your identity will be protected to the extent allowed by law. You will not be personally identified in any reports or publications that may result from this study.

2. The Department of Health and Human Service (HHS), other federal agencies as necessary, the University of Nevada, Reno Social Behavioral Institutional Review Board may inspect the data collected during this research project.

3. You will not provide your name, and it will never be known by your Instructor. However, it will be known to the person collecting the research data, Dr. Frederick Harris so that he can manage the disbursement of the food gift card incentives. As soon as the food gift cards have been disbursed, all data resources containing participant names will be deleted. In addition, once the research has been completed, the WebCampus survey itself will be removed.

4. Servers housing survey applications record and collect incoming IP addresses for system administration and record keeping. These data are analyzed only in aggregate; no connection is made between participants and their computers' IP addresses. These servers also use cookies to recognize visitors and more quickly provide personalized content, grant unimpeded access to the website, and to track usage behavior and compile data, in aggregate form only, for website improvement purposes.

5. You may close your Internet browser immediately after completing the survey to limit access to your survey responses, especially if you are using a computer in a public domain. If, after exiting the survey, you wish to remove the cookies from a personal computer, you may obtain instructions for deleting cookies from the help menu or with assistance from your Internet provider.

COSTS/COMPENSATION

1. There will be no cost to you for participating in this research study.

2. As an incentive to participate in the research study, you will be given a $5.00 coupon for food at Cantina Del Lobo at the Joe Crowley Student Union.

3. In addition, you will be entered into random drawings for $25.00 food coupons for the Cantina Del Lobo restaurant. The odds of winning the $25.00 coupons are fixed at 1 in 5.

DISCLOSURE OF FINANCIAL INTERESTS

The researcher involved with this study has no conflict of interest that would bias the study, and will not gain or lose any financial benefit as a result of any of the possible outcomes of the study.

RIGHT TO REFUSE OR WITHDRAW

1. You may refuse to participate or withdraw from the research study at any time by simply closing the WebCampus survey item.

2. You may skip questions that you do not want to answer.

3. If the study design or use of the data is to be changed, you will be so informed and your consent re-obtained.

4. You will be told of any significant new findings developed during the course of this study, which may relate to your willingness to continue participation.

RESEARCHER CONTACT INFORMATION

If you have questions about this study, please contact Michael Leverington at (775) 784-1414 or Dr. Frederick Harris at (775) 784-6571 at any time.

PARTICIPANT RIGHTS CONTACT INFORMATION

You may ask about your rights as a research subject or you may report (anonymously if you so choose) any comments, concerns, or complaints to the University of Nevada, Reno Social Behavioral Institutional Review Board, telephone number (775) 327-2368, or by addressing a letter to the Chair of the Board, c/o UNR Office of Human Research Protection, 205 Ross Hall / 331, University of Nevada, Reno, Reno, Nevada, 89557.

O	1. I voluntarily consent to participate in this research study.
O	2. I do not wish to participate in this research study.

2. USE of the Five Step Programming Process

How much do you think you used the Five Step Programming Process for working out your most recent laboratory program?

a. 0%
b. 20%
c. 40%
d. 60%
e. 80%
f. 100%

3. DEVELOPMENT using the Five Step Programming Process

How much do you think the Five Step Programming Process helped you develop your most recent laboratory program?

a. 0%
b. 20%
c. 40%
d. 60%
e. 80%
f. 100%

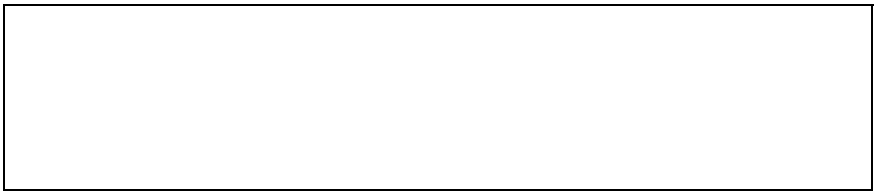
4. TIME TAKEN using the Five Step Programming Process

Do you think that using the Five Step Programming Process allowed you to complete your most recent laboratory program in more, less, or about the same amount of time?

a. it helped me complete the program in a lot less time
b. it helped me complete the program a little less time
c. it didn't make a difference with my program completion
d. it caused me to take a little more time to complete my program
e. it caused me to take a lot more time to complete my program

5. DIFFICULTIES with the Five Step Programming Process

Did you have any difficulties when using the Five Step Programming Process for developing your most recent program? If you did, please explain briefly.



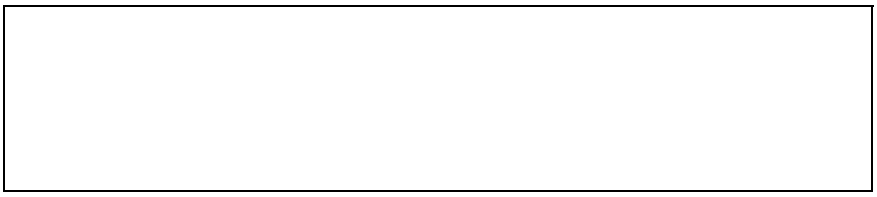
6. LEARNING with the Five Step Programming Process

How much do you think that using the Five Step Programming Process helped or hindered your learning in this programming course so far?

a. it helped me learn about programming a lot
b. it helped me learn about programming a little
c. it didn't make a difference with my learning to program
d. it was not very helpful with my learning to program
e. it was a hindrance to my learning to program

7. USING the Five Step Programming Process FOR OTHER COURSES

Do you think using the Five Step Programming Process might help with other courses, such as math, science, etc? Please explain briefly.

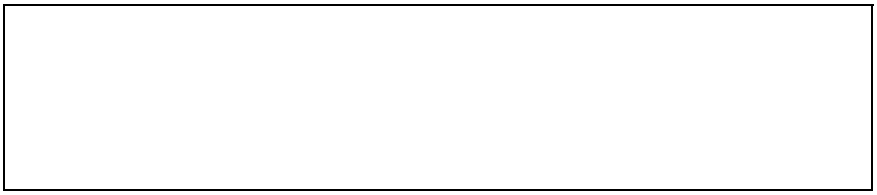


8. CONDENSING DOWN PARTS of the Five Step Programming Process

Consider the brief description of each of the five steps below:

Step 1: very brief program overview, written in comments
Step 2: extended program overview, written in comments
Step 3: program overview with function specifications, written in comments
Step 4: main program code developed, other functions prototyped and stubbed
Step 5: whole program completed

Are there parts of the five steps you think should be condensed down? For example, because step 1 is so brief, do you think it would help to combine steps 1 and 2? Please answer with a brief explanation below.

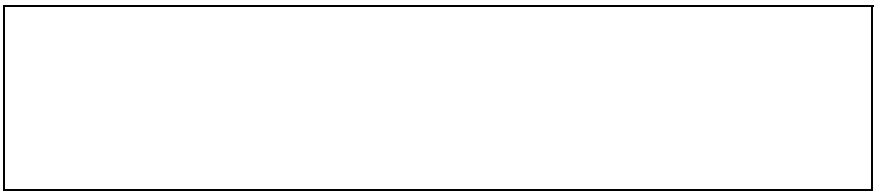


9. EXPANDING OUT PARTS of the Five Step Programming Process

Consider the brief description of each of the five steps below:

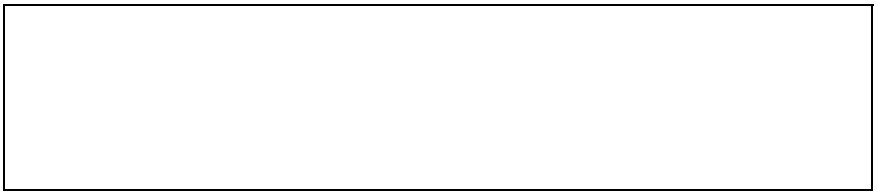
Step 1: very brief program overview, written in comments
Step 2: extended program overview, written in comments
Step 3: program overview with function specifications, written in comments
Step 4: main program code developed, other functions prototyped and stubbed
Step 5: whole program completed

Are there parts of the five steps that you think should be expanded upon? For example, because you write both the main program and the stub function comments in Step 4, do you think it would help to break this process into two unique steps? Please answer with a brief explanation below.



10. ANY OTHER ISSUES with the Five Step Programming Process

If you have any other opinions or thoughts on your use of the Five Step Programming Process, please share them here.



Appendix B - Complete Data Set

The following data set constitutes all the information collected from the students. The written comment data is organized in groups as explained in Chapter 5.

Q 1		
Usage		
0.00%	1	3.85%
20.00%	6	23.08%
40.00%	1	3.85%
60.00%	5	19.23%
80.00%	10	38.46%
100.00%	3	11.54%
sum:	26	

Q2		
Development		
0.00%	5	19.23%
20.00%	3	11.54%
40.00%	3	11.54%
60.00%	2	7.69%
80.00%	9	34.62%
100.00%	4	15.38%
sum:	26	

Q3		
Time Taken		
lot more	2	7.69%
little more	8	30.77%
no difference	3	11.54%
little less	10	38.46%
lot less	3	11.54%
sum:	26	

Q5		
Learning Help		
hindrance	1	3.85%
not helpful	2	7.69%
no difference	4	15.38%
helped a little	13	50.00%
helped a lot	6	23.08%
sum:	26	

Q4 Difficulties

NONE

No

No.

no

none

No, it pretty self-explanatory.

No difficulties. I think the 5 step programming process helps with a rough outline of the program but not beyond that detail level.

No, not at all. I have had problems in the past but it was more due to unfamiliarity with the code concepts, not the process.

No problems, it's pretty straight forward.

The Five Step Programming process is necessary for developing large programs. I didn't have any difficulty using it in my last program, and have had very little trouble using it in programs past. My only issue with the five step programming process is with its description on WebCampus: the directions should be a little more clear, such as in Step 3, where the only explanation used is an example.

GOT BETTER LATER

Yes, at the beginnig of the semester the Five Step Programming Process was a waste of time, it caused me to take a lot more time to complete my program. But at the opposite I think it helped me do complete my program a little bit faster for the last two labs.

Some times it seems like it generated more busy work when the programs were simple but when they became more complex it really helped

HARD TO COMPLETE THE PROCESS

I found that using the five step programming system helped in some areas but hindered others, because i would forget areas of the program that needed to be implented until the final step so i would have to back and add code in.

When you're adding new information to an assignment based on the 5 step programming process design assignment, or if the design assignment was incomplete, it's discouraging and more time consuming having to edit additional information related to the programming assignment

The fourth step almost never translates well into the fifth step. Often, coding that I am unfamiliar with or not acquanted will cause massive errors in my program that force me to restructure it.

WANT TO CODE FIRST

I always want to start writing out the code while commenting it out because I feel like I'll forget my idea later.

I wouldn't say that the 5 step process is diffucult, it is just faster and easier for me to work the other way around (writing the functions first).

The 5 step process is not the best way to write the program. Writing down what one plans to do to complete the entire program before writing code is a bad idea. First, if and when there is a problem the programmer must look at pages of unfinished code to find the problem. If one instead writs a small test program and starts with a getting one function working than going from there, problems can be easily found and fixed.

DIFFICULT TO DO OR UNDERSTAND

It's a good concept, but it's difficult to actually do and it doesn't always help.

sometimes your brain would naturally take you off course and it would be hard to get your mind back into the way the five step process wanted you to do it.

It is more difficult for me to think from a top down manner than from a tool use method. I would rather build the smaller components before and then utilize the working tools then try to do an overall and guess at what I will need.

It is good as a framework of developing the porgram (the thought process), but tyring to adhere to it in developing the actual program has caused me to mess up the logic and unable to complete my assignment properly.

step 5 is hard

NO POSITION

I didn't understand the process at first. Dr. Louis did not cover it in class, so maybe next time the process should be covered in the Lab. That would be much more beneficial, I think.

Q6 Using 5-S for other courses

HELPFUL ELSEWHERE

I think the methodology behind it, taking a big problem and splitting that problem into little parts, is uesable whenever you want to solve any problem.

Yes, I believe it's applicable to other courses and I have used it in other courses for projects and essays.

MAYBE HELPFUL ELSEWHERE

Not sure, but it is definitely a great process for coding. As coding can require intense problem solving it was really helpful do document everything in small chunks so it was easier to digest. When it comes to math, I think it is much easier to solve equations and problems without having to plan things this thoroughly.

I believe that it would help other subjects because it teaches a person to build a foundation and understand the smallest component of something before going on to the more complex information

The program could really be used for any other class. It is basically saying what you should do in an assignment then actually doing it.

It's always good to have a strategy or plan of attack for solving problems. If the problem is complicated and takes several steps to complete, the five step programming process would be very effective. For math, there's usually one formula or way to solve an equation. In programming, there is an infinite number of ways to being coding a program. Oftentimes, overthinking your program can be counterproductive. You're less likely to overthink stoichiometry or an integral.

Sure. Basically the 5 Step Programming process is an outline, but it might be more trouble that it's worth for other subjects, as other subjects' problems tend to be more trivial.

I don't think so. Maybe witting essay, but who doesn't already plan out there essay before witting them?

The process itself seems to be related to only computer science; however, the logic behind it definitely carries over into courses of other fields such as mathematics and physics. The idea of finding the main goal of a problem and breaking it down into smaller problems is essential to both this course and others.

It has possible uses dependant on each students learning ability.

Possibly for really complex problems for calculus. I've actually never really thought about it.

I don't think it will help me in courses other than computer science, but it is possible that it could and I just don't know it yet.

It is a systematic thinking process that can be learned in other ways, esp. thorugh other subjects. I think the concept is good, however, the way the class is designed made this process a little too rigid to be useful. In fact, it is easier for me to write my entrie program and remove my codes in order to complete the "Design Assignment" part of the homework.

Yes, the idea of moving from the general to the specific and breaking problems up into smaller pieces I can see helping in other courses.

maybe

Yes i think it will help in assignment such a designing big project. For my major (Civil engineering) it is usefull to proceed as the Five Step Programming Process, consider the biggest, and then move on the smallest part.

DON'T THINK SO

I do not think the process will be helpful with other courses in its specific structure. I think it does help to teach how to break a problem down into smaller chunks.

no For math, one would not write down pages of commentary on how they thought they were going to solve the problem before attempting the problem.

No, I dont think I will. I dont know how it would apply to my other classes

No, I find it easier to piece small pieces together to understand something larger rather than breaking down large things into smaller ones.

No

No, it is more or less a step made for planning, which I could do just as well without this five-step process. In fact, I would find it a large bother to translate its application to something other than coding, given my other subjects are more about learning specific equivalents of functions rather than a large algorithm.

not really in math, but it could be helpful to developing an idea in a science.

no, I won't be taking other courses like that

No i dont think it would be useful. Since most other courses do not solve problems in a manner that can be modularized.

Q7 Condensing Down

OKAY AS IT IS

I like keeping the steps individually, as taken one step at a time it develops very well.

It's great the way it is because of the way each step builds on the previous one.

step one and step two should be different becuase it helps to design the program as a whole.

I dont believe that any of the steps should be combined or condensed down. Every step helps the user understand what is needed for programming

N. -- assume this means no --

No, it seems properly condensed.

No. 5 is a nice number.

Having 5 steps is a good way to go, mainly because it helps the student focus on the problems at hand. Stepping through each large problem, and solving smaller problems to accomplish it relieves a lot of stress. Also, the feeling of accomplishment after each step acts as a great motivator to continue through the rest of the program.

No, because a 5 step program with only 3 steps would be misleading.

No, I think it's all necessary to the development of the process.

I think it's fine the way it is. Each step adds just the right amount of information to the process.

CONDENSE STEP 1/2

I think consolidating 1 and 2 would be helpful because if anyone else is like me, they don't remember the specifications of all five steps. I pretty much know what step 4 is supposed to look like from the design assignments.

Step 1 & 2 are pretty much the same, so they should be condensed together.

Combining steps 1 and 2 would be helpful in that the steps would all be very distinct; it would be easier to follow if you knew exactly what step you're on. In my mind the line between step 1 and 2 is not clear.

Steps one and two could be easily merged. I sometimes even did the first three all at the same time.

Maybe the step one and two can be combined...

CONDENSE STEP 1/2/3

In my personal execution of the process, I would almost always start at step 3/4. The reason is because, in my opinion, the first 2 steps outline things that I don't really feel are too confusing or complex to need commenting. (i.e. basic understood functions, obvious coding needed in main()). However, steps 3 and 4 are great in helping with the complexity of the code that can arise inside functions. (i.e. loop iteration, array augmentation, etc.)

I think steps 1, 2, and 3 can be condensed down into one step. In the beginning, it helped a lot to be able to do 5 steps but as the semester carries on, step one and two become vestigial steps.

I would have only three steps Step 1: Think of a function that will be needed. Step 2: Write the function. Step 3: Test the function.

REMOVE STEP 3

Step 3: function specifications. I'm usually only guessing what my functions are or what they can do. To base my program in step 4 around these functions can be bad. I'm a think as I go person, planning things out too far ahead usually results in bad work. I would cut step 3 out, and just go from step 2 to step 4.

CONDENSE STEP 4/5

The only complaint that I have about the 5 step process is step 4 and 5 should be condensed because often times I find that I need to adjust or modify my main program code when I am writing my functions which can often render my step 4 work useless.

CONDENSE ALL

I never do this step on at a time. I think it would be ridiculous to do them one at a time. So yes I think they could be condensed down.

outline work on the functions and what they are doing place the funcitons where they need to go comment the functions finish everything

DON'T USE STEPS

I like to break down problems as I see fit for each individual problem, it is a waste of time to have a single approach that you try to use for every problem since many problems are handled differently

Q8 Expanding out

OKAY AS IT IS

No

No.

No

I think 5 steps is plenty.

No.

no

Not really. I like the number 5.

No, I think it is expanded enough as it is.

No.

I think it's fine the way it is.

I think the expanding is not beneficial at this level of CS

No, I think step 3 and 4 are great as they are, because they help you get prepared for coding in a fairly concise manner, and I see no reason to arbitrarily expand on the process.

I think consolidation is more beneficial than expansion.

Commenting out the stub functions in step 4 helps keep track of what the main program is supposed to do; therefore, I believe it should be left the same.

Instructionis for step one should emphasize the importance of keeping it very basic, other than that the process worked very well for me.

BREAK DOWN STEP FOUR

There is a lot in step 4. Breaking it up would help.

The only aspect that could be elaborated on would be step 4. I found it difficult to know what exactly i wanted a function to do and how it was going to be implemented.

Step four is very big, and should be break down in multiple parts

OTHER

I do think that stubbing out the functions should be step 4 and writing main should be added to step 5.

Step 2 should include that it is a form of step 1 applied to segments and then reapplied until the function is split into small enough pieces.

for all steps make it work and have your ideas formulated

See above
answer.

It would help to actually write the functions instead of just stubbing them. Sometimes, it would help wirting the main program.

yes

Q9 Other Comments

NO ISSUES

I have no other issues with the five step process.

No.

I dont have any other issues

No.

LIKED IT

I enjoyed it, however, i found that sometimes it was more of an annoyance than help.

It's a good process, but it could use a better description online. The steps listed above in this survey is a nice summary. Put that online.

it can help but it can take a while to get your brain to think in computer terms and what it really going on, sometimes your can over simplify something and miss an easier way

This process is a very powerful tool, and is helpful in writing efficient code. The only suggestion I have is to introduce the process earlier in the course, saving students much frustration toward the class, and even the instructor :)

STARTED OUT ROUGH, GOT BETTER

Originally I hated having to use the programming process as I felt it made me spend extra time outlining when I didn't need to. This was because I hadn't really encountered anything in class that was really of any conceptual difficulty. When we started getting into loop, array, and file streaming design, the use of the process really became more important in my programming. Once I embraced actually taking the time to outline everything it really helped me conceptualize code structure and potential problems a whole lot easier. I really feel like a much stronger coder now, and I am very glad that I came around to using the programming process. CS 135 was easily my most time consuming class this semester, but because I feel that I learned so much, it was far and away my favorite. It also helps greatly that Michael is a very approachable and extremely competent instructor who does a great job of outlining classroom concepts.

DIDN'T LIKE IT, BUT IT HELPED

The process is tediuous, and certainly not something I enjoy doing, however it allowed my programs to come out looking relatively clean and usually I understood more of my program coming out than going in.

DIDN'T LIKE IT

The general consensus in our lab (just going off overheard comments) is that the process is not particularly helpful. Many students don't use their design assignments at all when completing their programming assignments. I use mine, but I don't follow the process to complete the design assignment, I just make it follow the format in the end.

People’s minds work differently. All students should not be required to follow the same formula. If the students understands what is needed to be done to achieve a goal (writing a program), they should not be forced to do it a certain way. The students should be shown different methods and be allowed to chose which method works best. CS 135 would be a better class if the students were only graded on the programming assignment and not the design assignment.

For a beginner to programming, the five step process is solid. It makes readable, easy to follow code for you and other users. Later on, I would be more concerned with finishing a programming assignment rather than completing a design assignment that may not accomplish my original goals, and therefore be scrapped. I'm essentially writing my thoughts down in english with the 5 step programming process. If I'm doing that, why can't I just write down my thoughts in english outside of the program?

I found actually coding a skeleton of the main program while commenting what I was doing was most effective. From there, I would add on flags, additional features, and the like. Even while doing a design assignment and no actually coding, I still confused myself with all the parts going on. I think having a partial program running helped me see what I did and did not have, which allowed me to handle the next task without thinking about how it interacted with every other task. I'll point out that when just planning, I was not sure if all the parts would mesh correctly which led to a lot of mental stress over handling the seperate parts appropriately. Step 1 should take care of this, but it did not. Finally, one program I wrote out the entire design assignment, then could not follow it when I had to code it. Instead, I simultaneously coded and commented it. When I compared the comments, they were almost exactly the same.

SHOULD BE OPTIONAL

It should be
optional

I think it should be optional. I don't see the need for this tool, in fact I find my self witting the code as I do it anyway, going back commenting on it then deleting the code I wrote. this seems very counter productive. I guess it might help some people but I really had no use for it.

SUGGESTIONS FOR IMPROVED TEACHING

Like I said, the programming process should be covered in lab one week, instead of class.

Although code is not generally allowed, I believe code up to step 4 should be allowed to be shared among students. The code is not at completion, so the students can still struggle learning but in an easier fashion. It is the equivalent of helping a person with code orally except it is on the screen.

You should introduce the Five Step Programming Process slowler. Probably make the students to write step one the first week, and release the programming file on friday with step one through five written by the teacher Then make the student write the step 1 and 2 for the second assignment, etc.

NOT RELEVANT

As a grading criteria, it was acceptable, though sometimes vague on the grading ruberic. It seemed to get 20 points I only need to spend 20 minutes typing random stuff, to get the last 5 points took several hours of reasoning and trial and error. While I appreciate the free points, it was only the last 5 points that accomplished the task of the 5 step Programming Process.

Some times I feel like I'm doing it incorrectly.

lab is hard

Appendix C - IRB Approval Letter

University of Nevada, Reno

Office of Human Research Protection
205 Ross Hall / 331, Reno, Nevada 89557
775.327.2368 / 775.327.2369 fax
www.unr.edu/ohrp

Certification of Approval
Social Behavioral, Panel B
Institutional Review Board

Date: April 15, 2010

To: Frederick Harris, PhD
Department of Computer Science and Engineering / 0171

CC: Michael Leverington
Department of Educational Specialties / 0432

UNR Protocol Number:	**SB09/10-122**
Protocol Title:	**The Five Step Programming Process**
Sponsor Protocol Number:	
Sponsor:	**Personal Funds**
VA Research:	**No**
UNR Assurance Number:	**FWA00002306**
IRB Number:	**IRB00003498**
Action Item:	**New Protocol: Social Behavioral**
Level of Review for Action:	**Expedited**
Expedited Category:	**7**
Review Period:	**12 months**
Final Approval Date:	**April 15, 2010**
IRB Action Date:	**April 13, 2010**
Expiration Date:	**April 13, 2011**

This approval is for:
Protocol application, as revised, 04/14/10
Recruitment verbal script, as revised, 04/14/10
Waiver or alteration of consent: Online information sheet, as revised, 14 April 2010
Research survey, as revised, 04/14/10

PI responsibilities

- Continuing projects must be reviewed and approved prior to the expiration date.
- Proposed changes must be reviewed and approved by the IRB prior to initiation, except when necessary to eliminate apparent immediate hazards to subjects. Such exceptions must be reported to the IRB at once.
- Any unanticipated problems which may increase risks to human subjects or unanticipated adverse events must be reported to the IRB within 10 days of becoming aware of the issue.
- When the project has been completed, please submit a closure request to the IRB.

Please reference the protocol number above on all related correspondence with the IRB. If you have any questions, please contact Valerie Smith at 775.327.2368.

Chair, Vice-Chair, or OHRP Director
Susan Ford Publicover, MA, CIP, Director

Cert_exp