

From Novice to Expert: Harnessing the Stages of Expertise Development in the Online World

Douglas A. Kranch
Professor, Computer Information Science
North Central State College
2441 Kenwood Circle
Mansfield, OH 44906
(419) 755-4788
dkranch@ncstatecollege.edu

Abstract

Expertise develops in three stages. In the first stage, novices focus on the superficial and knowledge is poorly organized. During the end of the second stage, students mimic the instructor's mastery of the domain. In the final stage, true experts make the domain their own by reworking their knowledge to meet the personal demands that the domain makes of them. Thus, as expertise develops, learning shifts from acquiring surface knowledge to constructing deep knowledge. Using the teaching of computer programming to exemplify the techniques, types of online and face-to-face learning experiences are discussed that are appropriate as expertise grows and learners gradually shift from making sense of the simple surface features of the domain to acquiring experience through increasingly complex problems that deepen learning.

Introduction

Teaching programming to novices has been seen as a problem for decades (Bennedsen & Caspersen, 2008) and been regarded by some as one of the seven grand challenges of computing (McGettrick, Boyle, Ibbett, Lloyd, Lovegrove, & Mander, 2005). In a well-meaning attempt to bring novices "up to speed" quickly and bypass the frustrations they encountered on their way to expertise, experts may begin introductory programming courses by showing novices the various "tricks of the trade" they have learned or discuss the broad principles of structured vs. unstructured programming. The results are often disappointing as students seem unable to grasp what are to instructors the simplest programming principles at the end of the course (McCracken, Almstrum, Diaz, Guzdial, Hagan, et al., 2001). This paper proposes a theory of expertise development that sheds light on why this communication gap occurs between expert and novice programmers and provides techniques that can be used to bridge that gap.

Expert Mental Organization

Ericsson and Lehmann (1996) define expert performance as "consistently superior performance on a specified set of representative tasks for a domain" (p. 277) by acquiring a set of skills, knowledge, and perceptions that helps them overcome specific critical limits. Yet this improved performance is the product of a working memory that according to Cowan (2000) can attend to just four independent ele-

2009 ASCUE Proceedings

ments simultaneously. For expertise to develop, more information must be compressed into those four elements to allow capacity for manipulating that information.

Sweller and Chandler (1994) asserted that the principle means of reducing working memory load available to the mind are schema acquisition and automatization. Schemas reduce working memory load by combining simple ideas into more complex ones in long-term memory, where they gradually begin executing automatically (van Merriënboer & Sweller, 2005) and no longer require any working memory.

Dreyfus and Dreyfus (1986) described expertise as a process that moved problem solving from conscious analytical thinking to intuition. They characterized the novice as learning calculations and heuristics and following them exactly without exception. Experts on the other hand lose consciousness not only of how in detail they perceive the situation (the perception becomes simply a feeling) but also of the performance needed to react to it. An expert performer is immersed within the performance and responds smoothly and intuitively. When faced with extraordinary situations or unexpected results from their actions, experts revert to analytical problem solving by employing what they called “deliberative rationality” (p. 36) that evaluates their intuitive responses in a search for better ones.

Zeitz and Spoehr (1989) compared breadth first and depth first learning and concluded that the mental organization of their learners went through three developmental stages to reach the level of expertise. In the first stage, novices focus on superficial characteristics as they train their perceptual abilities. The arrangement and order in which those perceptions are presented eventually have a profound effect on mental organization. In this early stage, the few knowledge chunks novices acquire are generally small, disconnected, poorly organized, and centered on surface characteristics. The domain seems overwhelmingly complex and learners grasp for hooks that relate what they are learning to already-established knowledge.

It is during the second stage that learners wean themselves from depending on previously-mastered knowledge to cope with the new domain and assimilate its knowledge in an “orderly, hierarchical fashion” (p. 328). They can explain how they use its knowledge and skills to solve problems. Since learners tend to organize knowledge in the way it has been taught, one would also expect that their organization of the domain would mimic the instructor’s. During the third stage of expertise, continued application of learning to real-world problems and the unique demands those problems make uniquely reworks the domain’s cognitive organization and produces complex, personalized expertise. As Dreyfus and Dreyfus (1986) described, experts respond fluidly to the demands of the domain, and they do this by reworking their knowledge to meet the unique demands that the domain has made on them.

Developing Programming Expertise

Computer programming demands complex thinking and creativity, and teaching it has been a continuing challenge to educators. McGettrick et al (2005) cite the effective teaching of programming knowledge and skills as one of the grand challenges of computing. Programming novices will tend to focus first on the syntax (or grammar) of the language of choice and try to program by rules. They examine the language’s surface characteristics in the general belief that programming code that looks the same will act the same. Novices have incomplete and poorly recalled chunks of knowledge with sizeable gaps in their overall conceptual organization of the programming domain. This fragmented knowledge and lack of

2009 ASCUE Proceedings

organization will be reflected in every programming activity in which they engage. Experts by comparison possess well developed and uniquely organized knowledge and skills that evince complexity and allow them to perceive quickly the elements of the problem that are critical to the success of their programs.

If the development of expertise fundamentally alters the expert's mental conception of the domain to the point that thinking becomes so intuitive that experts struggle to explain how they solve problems, then instructors who are experts in their fields will find it nearly impossible to teach their way of thinking to novices. The principle reason why the presentation of domain knowledge from an expert's point of view can be unintelligible to novices is rooted in this change of mental organization. Expertise cannot be taught; rather, novices must themselves engage in experiences which develop the basic principles of the domain and allow their conceptions to first mimic those of the instructor, then evolve into unique domain conceptions that are shaped by their own successes and failures. As expertise develops, learning slowly shifts from acquiring surface knowledge to constructing deep knowledge.

One instructional model that supports the gradual development of expertise is Reigeluth and Stein's (1983) elaboration theory. It proposes that concepts be taught from simple to complex in an order that ensures all prerequisites are mastered as new topics are encountered. The theory centers on the *epitome*, a concrete application that presents a small number of the essential ideas related to a single type of course content (concepts, procedures, or principles). It advocates teaching all the required knowledge or skills together from the beginning and gradually elaborating on them in a spiral fashion rather than introducing concepts individually and amalgamating them toward the end of instruction.

Applying elaboration theory to Zeitz and Spoehr's (1989) stages of expertise, novices should begin with the basic syntax of the language of choice tied together with simplified semantics. Novices will learn the surface characteristics of the language and master bits and pieces of syntax and semantics (The two of which at this stage will seem to novices as essentially the same thing). Once simple syntax and semantics are mastered, more complex program plans and the semantics (or logic rules) of the language apart from the syntax can be introduced. As expertise grows and the learners master programming as taught by the instructor, so, too, should the complexity of the plans that are discussed and the problems that are solved.

Achieving the competence required during the second stage of expertise growth is no simple matter. Learners must make the sizeable leap of understanding the difference between the everyday use of logic (natural logic) and the formal logic employed in programming. While novice programmers do not enter their studies of programming already knowing a computer language, they do enter with years of experience using language and reasoning for everyday problems. Wood (1998) discussed the difference between natural and formal logic, noting that many unsaid implications are often drawn from the natural logic embedded in everyday language that could not be carried over into language employing formal logic.

Learners may have prior experience in solving problems and in writing those solutions down, but the formal logic of computer programming makes previous informal problem-solving experience at best misleading and at worst irrelevant to developing programming solutions. Inevitably, confusion between

2009 ASCUE Proceedings

their own natural logic and the logic of the programming language produces mistakes (Bonar & Soloway, 1983; Spohrer & Soloway, 1988).

Expert programmers with their unique and holistic knowledge base depend more on a top-down forward design methodology. They tend to retrieve comprehensive design plans from memory and construct high level mental models from them before working on the details (Ericsson & Lehmann, 1996; Rist, 1989; Soloway, 1986). Plans that make up the expert programmer's knowledge structure are unique not only to the experience that the programmer has had, but also apparently even to the kind of programming language in which the programmer was trained (Davies, 1993). As with Dreyfus and Dreyfus' (1986) experts caught in an extraordinary situation, only when a plan must be created from scratch to fulfill a specific program goal does a programming expert use a bottom-up "goal decomposition and plan recombination" (Rist, 1989, p. 403) strategy.

Mastery of the domain's problem-solving logic it seems must be learned through experience, not through explanation. Soloway (1986), for example, advocated teaching novices to mimic the top-down problem-solving method experts tend to use by having novices break down problems into segments small enough to match stock solutions they would be taught and then combine those solutions into a single comprehensive plan. When Mann (1991) applied Soloway's (1986) strategy, he found that students considered such stock solutions impractical and rigid, and thus not useful. The reason is plain: generic plans are what experts use to solve programming problems. Novices can only apply plans given to them in a means-end manner since they cannot incorporate into their thinking what they have merely memorized and not personalized. It is low-performing novice students who believe that applying memorized algorithms is a key to successful programming (Vodounon, 2006).

Learning Programming in the Online Environment

If immediately instructing novices in expert ways of programming is not productive, what techniques would work, especially in the online environment? Rather than beginning with the templates, plans, or strategic overviews experts use, instructors should first train novices on the syntax of the language and only after they can use it with some facility encourage them to develop a personal stock of plans for solving problems. Cognitive load theory tells us the material should be presented in a way that maintains manageable complexity: neither so easy that interest wanes nor so difficult that learners feel hopeless. Three types of cognitive load have been identified in the literature: intrinsic, extrinsic, and germane (Schnotz & Kürschner, 2007).

Intrinsic load is a measure of the minimum number of elements that must be held in working memory for a concept to be understood. It is not the same as task difficulty and is, in fact, almost completely independent from it (Chandler & Sweller, 1996). Learning a single element may be very difficult but cause little cognitive load, while attempting to juggle multiple simple elements in the mind's eye simultaneously could produce high cognitive load. This load can be reduced in several ways for which the online environment is highly suitable. Here are some examples:

- Information should be introduced only as it is needed by learners so that unnecessary information is not needlessly filling working memory (van Merriënboer, Kirschner, & Kester, 2003).

2009 ASCUE Proceedings

Examples: Methods for breaking a program into modules can wait until novices are writing programs large and complex enough to need the technique. New terms and procedures can be given links to embedded definitions, explanations, and examples that require only the information and concepts that have been introduced. They can be opened as needed by individual learners to provide the immediacy.

- Individual, elementary programming elements should be studied separately first, and only after they are mastered should groups of elements be studied together to form a single solution (Moss, Kotovsky, & Cagan, 2006).

Examples: Learners can be directed first to lessons that teach elementary programming elements, conditional statements (`if...then...else`) and loops for example, and when mastered as determined by unit assessments, groups of elements (e.g., loops containing conditionals) can be presented in a single lesson. In the same way, syntactic rules can be introduced in their most simplified form and these introductions linked to code statements in future examples that show elaborations of those rules.

- Goal-free tasks that allow learners to master tasks at their own pace such as worked examples, completion tasks, and reverse tasks that start from the answer and work back to the question should be used whenever possible (Sweller, 1994). Such tasks let the novice use scarce working memory solely to learn the task rather than have attention drawn to a goal. This relieves the means-end drive to solve a specific problem at the expense of gaining understanding.

Examples: Pages can be provided containing goal-free, “sandbox” problems that allow learners to experiment with the programming structures and their parameters. For example, novices can be encouraged to experiment with loops to see how they work rather than asked to write a specific loop that displays the word “Hello” ten times. Examples that let students fill in values into variables can build an understanding of how the structures work while removing the possibility of making syntax errors that can interfere with learning.

- Problems and examples should start simple and grow gradually more complex as learner expertise results in decreased cognitive load. Learners should be shown a variety of worked solutions to programs and observe experts as they solve realistic problems.

Example: Canned screenshot presentations can demonstrate expert problem solving of a variety of worked solutions, allowing learners to stop and replay the solutions at will.

- Practice should be distributed in bursts throughout the learning. While a few intense periods of massed practice can produce short-term recall, better long-term retention occurs when intrinsic load is reduced by well distributed practice (Fishman, Keller, & Atkinson, 1968).

Examples: A short exercise or two at the end of each single-topic Web page can help learners apply the new concept and result in long-term retention. End-of-unit exercises can then reinforce the learning from the topic-level exercises. These can be linked to definitions and concept explanations for point-of-learning review as needed by the learners.

2009 ASCUE Proceedings

- Guiding novices in their learning is more effective than asking them to determine for themselves what to explore (Tuovinen & Sweller, 1999).

Examples: Guidance is especially important in the online environment, where the emphasis on self-governance and the lack of personal contact with an instructor make the need for self-guiding lessons critical. Completion problems can be provided with links to a network of definitions and tutorials that learners can choose to use to help them diagnose their errors.

Extrinsic load is effort that results from the way the material is presented, its context rather than its content. It can be reduced by removing irrelevant material, thus reducing effort unrelated to learning (Schnotz and Kürschner, 2007). Sweller (1994) speaks to the unintended increase in extrinsic load by techniques often employed in online materials. The first is the use of illustrations. Sweller notes a split-attention effect that significantly reduces extrinsic load when explanatory text is worked into an illustration rather than placed into a separate block of text. The second is the use of repetition. Contrary to what may seem good instructional practice, Sweller (1994) found that including text, diagram, pictures, or other materials that present exactly the same information forces learners to integrate the different media into a single block of knowledge without any material gain in understanding. Instead, the increased extrinsic load resulted in decreased learning. An example of this often used in computer based instruction is reading text to learners while showing it simultaneously on the screen. A better approach would be to use audio to complement rather than repeat the visual information, such as a screenshot recording that uses voice to explain actions that are visible on the screen.

Germane load is the effort expended in building mental structures and automating learning. It can be varied, but the total cognitive load (the intrinsic, extrinsic, and germane loads added together) must not exceed the limit of the learner's working memory. Germane load is an important variable that can be adjusted to align learner expertise to the learning task and keep total cognitive load from being either too high or too low. Having novices rate the difficulty of an assignment using a 9-point Likert-like scale with categories ranging from "very, very easy" to "very, very difficult" can help quantify the germane load (DeLeeuw & Mayer, 2008).

After novices have become sufficiently comfortable working with the language syntax, they should be encouraged to develop their own library of programming solution plans. As noted above, searching for a plan that matches the demands of a given situation is one of the expert's ways of solving programming problems. They may find it easier to do this if instructors first help them to find the focus line to epitomize a piece of code, then gradually expand to groups of focal lines, and finally to see the entire code as a unified solution (Rist, 1989).

The order in which program semantics is introduced also should be set so that the simpler and clearer semantic rules are presented before the more confusing and mentally taxing ones. For example, understanding the difference between iteration and recursion can be a difficult concept for novices. Kessler and Anderson (1986) studied this problem and found that the subjects could learn recursion as easily as they did iteration, but the order in which they learned them was critical. When participants learned recursion first, they simply memorized a set of statements and used a means-end strategy to find a set that solved the problem. When afterward asked to learn iteration, they began with as little understanding as participants who had received no previous instruction. Learning the iterative function first, on the other

2009 ASCUE Proceedings

hand, made transfer to a recursive problem faster since the iteration training gave some notion of program control. Thus, concepts that engender useable mental models of programs in general and flow of control in particular are essential to learning programming.

Novice programmers tend to come from backgrounds that have included computer use during a considerable part of their lives. We can therefore expect that methods they have learned to solve previous problems they have had with computers should be extended to programming. It is common to restart a computer that has stopped because of a faulty program; thus, novices should also be expected simply to recompile a program that does not compile the first time in an attempt to “reboot” it. Many novices need to be taught explicitly that the error is with their code, not the compiler (Simon, Bouvier, Tzu, Lewandowski, McCartney, & Sanders, 2008). Other instructional techniques that can enhance novice’s understanding of semantics include the following:

- Teach novices how to derive the function and output of a program from its code to help make the implicit relations in program code explicit (Corritore & Wiedenbeck, 1991).

Examples: Present blocks of code to students as online discussion questions that ask them to identify the purpose of the code to help sharpen their code deciphering skill. Links can also be embedded to brief presentations of program plan models to help forge a link between problem solving from a reservoir of strategic plans.

- Teach debugging skills to make the difference between locating and fixing errors clear. Only after experience with faulty programs will learners begin to look deeper into their own programs for the causes of faults. As they lay the cause of program faults more to syntax or semantic errors rather than elements out of their control, they will lower the number of total mistakes they make (Masuck, Alves-Foss, & Oman, 2008).

Examples: Code tracing exercises, clear and specific (rather than general) explanations of what program statements do, practice with basic programming elements until they are automatized, and emphasis on creating meaningful variable names and comments can develop these skills (McCauley, Fitzgerald, Lewandowski, Murphy, Simon, Thomas, & Zander, 2008; Simon et al, 2008). Presenting error detection exercises as discussion questions can increase the understanding of struggling learners by letting them see the responses of those who have better mastered the concepts.

Caveat: Having learners to work through program checking routines may prove valueless until they have advanced far enough to understand the processes they are tracing and debugging (Pawley, Ayres, Cooper, & Sweller, 2005).

- Give students opportunities to read code that exemplifies good practice. Novices will implicitly form rules of correct coding syntax and semantics from viewing good code and set up code patterns that can help check against poorly formed code (Servan-Schreiber & Anderson, 1990).

2009 ASCUE Proceedings

Example: A link on all pages to a library of code examples named to relate to specific concepts taught in the course will guide learners to the help they need when writing their own code. Links to worked examples that learners can modify will give them the opportunity to experiment with the language syntax and semantics.

- Novices should practice building program plans that summarize the key intentions of a program, and they should practice using program plans to reason from them what the program is intended to do (McCauley et al., 2008).

Examples: Discussion questions centered on specific plans can provide insight to instructors on where individual students are in this vital expertise indicator. They can also give struggling learners an opportunity to observe the thinking processes of more advanced learners. Successive completion problems patterned after specific programming plans that let learners add more code with each problem can provide scaffolding for learners as well as indicate which parts of the plans are the more difficult.

Even if they use these methods, experts can no longer trust themselves to gauge the cognitive load a given exercise will produce in novices. Having transformed clusters of concepts and skills that novices see as complex and vast into fewer, larger, concentrated chunks (Sweller & Chandler, 1994; Schnotz & Kürschner, 2007), what was once difficult as a novice is now done automatically (van Merriënboer & Sweller, 2005) and so seems almost effortless. Letting novices rate the mental effort they used to complete an assignment is a simple way to gain an accurate measure of the cognitive load they experience from the assignment (Paas & Van Merriënboer, 1994).

Conclusion

Expertise is not developed by the simple accretion of knowledge; rather it results from a complete reworking of the mental organization of a domain. By seeing it as such, experts who are also teachers can better appreciate the difficulty a new domain presents to novices and use the stages of expertise development as guides to developing and measuring expertise in novices. Novices become more proficient in using programming elements and their combinations as they grow in expertise, shifting their focus gradually from the surface features to the deep structures of programming. The learning experiences provided to novices must allow for this shift by using instructional techniques that expand, contract, and recycle through old material as needed. The online environment can be designed to respond to these individual learner needs and thereby help turn novices into experts more successfully.

References

- Bennedsen, J. & Caspersen, M. E. (2008) Optimists have more fun, but do they learn better? On the influence of emotional and social factors on learning introductory computer science, *Computer Science Education*, 18(1), 1 – 16.
- Bonar, J. & Soloway, E. (1983). Uncovering principles of novice programming. In *Proceedings of the 10th ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages* (pp. 10–13). New York: ACM.

2009 ASCUE Proceedings

- Chandler, P., & Sweller, J. (1996). Cognitive load while learning to use a computer program. *Applied Cognitive Psychology*, 10(2), 151-170.
- Corritore, C. L., & Wiedenbeck, S. (1991). What do novices learn during program comprehension? *International Journal of Human-Computer Interaction*, 3(2), 199-222.
- Cowan, N. (2000). The magical number 4 in short-term memory: A reconsideration of mental storage capacity. *Behavioral and Brain Sciences*, 24(1), 87-114.
- Davies, S. P. (1993). The structure and content of programming knowledge: Disentangling training and language effects in theories of skill development. *International Journal of Human-Computer Interaction*, 5(4), 325-346.
- DeLeeuw, K. E. & Mayer, R. E. (2008). A comparison of three measures of cognitive load: evidence for separable measures of intrinsic, extraneous, and germane load. *Journal of Educational Psychology*, 100(1), 223-234.
- Dreyfus, H., & Dreyfus, S. (1986). *Mind over machine: The power of human intuition and expertise in the era of the computer*. New York: Free Press.
- Ericsson, K.A., & Lehmann, A.C. (1996). Expert and exceptional performance: Evidence of maximal adaptation to task constraints. *Annual Review of Psychology*, 47, 273-305.
- Fishman, E. J., Keller, L., & Atkinson, R. C. (1968). Massed versus distributed practice in computerized spelling drills. *Journal of Educational Psychology*, 59(4), 290-296.
- Kessler, C.M., & Anderson, J. R. (1986). Learning flow of control: Recursive and iterative procedures. *Human-Computer Interaction*, 2(2), 135-166.
- Mann, L. M. (1991). The implications of functional and structural knowledge representations for novice programmers. *Dissertation Abstracts International*, 53(05), 1470. (UMI No. 9228760)
- Masuck, C., Alves-Foss, J., & Oman, P. (2008). Analysis of fault models for student use. *SIGCSE Bulletin*, 40(2), 79-83.
- McCauley, R., Fitzgerald, S., Lewandowski, G., Murphy, L., Simon, B., Thomas, L., & Zander, C. (2008). Debugging: a review of the literature from an educational perspective. *Computer Science Education*, 18(2), 67-92.
- McCracken, M., Almstrum, V., Diaz, D., Guzdial, M., Hagan, D., Kolikant, Y. B., Laxer, C., Thomas, L., Utting, I., and Wilusz, T. (2001). A multi-national, multi-institutional study of assessment of programming skills of first-year CS students. *SIGCSE Bulletin*. 33(4), 125-180.

2009 ASCUE Proceedings

- McGettrick, A, Boyle, R., Ibbett, R., Lloyd, J, Lovegrove, G, & Mander, K. (2005). Grand challenges in computing: Education – A summary. *The Computer Journal*, 48(1), 42-48.
- Moss, J., Kotovsky, K., & Cagan, J. (2006). The role of functionality in the mental representations of engineering students: Some differences in the early stages of expertise. *Cognitive Science*, 30(1), 65-93.
- Paas, F. G., & Van Merriënboer, J. (1994). Measurement of cognitive load in instructional research. *Perceptual & Motor Skills*, 79(1), 419-430.
- Pawley, D., Ayres, P., Cooper, M., & Sweller, J. (2005). Translating words into equations: A cognitive load theory approach. *Educational Psychology*, 25(1), 75-97.
- Reigeluth, C. M. & Stein, F. S. (1983). The elaboration theory of instruction. In C. M. Reigeluth (Ed.) *Instructional-design theories and models: An overview of their current status* (pp. 334-381). Hillsdale, NJ: Lawrence Erlbaum Associates.
- Rist, R. S. (1989) Schema creation in programming. *Cognitive Science*, 13, 389-414.
- Schnotz, W., & Kürschner, C. (2007). A reconsideration of cognitive load theory. *Educational Psychology Review*, 19(4), 469 - 508.
- Servan-Schreiber, E. & Anderson, J. R. (1990). Learning artificial grammars with competitive chunking. *Journal of Experimental Psychology: Learning, Memory, and Cognition*, 16(4), 592-608.
- Simon, B., Bouvier, D., Tzu, Y., Lewandowski, G, McCartney, R., & Sanders, K. (2008). Common sense computing (episode 4): debugging. *Computer Science Education*, 18(2), 117-133.
- Soloway, E. (1986). Learning to program = learning to construct mechanisms and explanations. *Communications of the ACM*, 29(9), 850-858.
- Spohrer, J. C. & Soloway, E. (1988). Novice mistakes: Are the folk wisdoms correct? *Communications of the ACM*, 29(7), 624-632.
- Sweller, J. (1994). Cognitive load theory, learning difficulty. and instructional design. *Learning & Instruction*, 4(4), 295-312.
- Sweller, J. & Chandler, P. (1994). Why some material is difficult to learn. *Cognition and Instruction*, 12(3), 185-233.
- Tuovinen, J. E., & Sweller, J. (1999). A comparison of cognitive load associated with discovery learning and worked examples. *Journal of Educational Psychology*, 91(2), 334-341.
- van Merriënboer, J. J. G., Kirschner, P. A., & Kester, L. (2003). Taking the load off a learner's mind: Instructional design for complex learning. *Educational Psychologist*, 38(1), 5-13.

2009 ASCUE Proceedings

van Merriënboer, J. J. G., & Sweller, J. (2005). Cognitive load theory and complex learning: Recent developments and future directions. *Educational Psychology Review*, 17(2), 147-177.

Vodounon, M. A. (2006). Exploring the relationship between modularization ability and performance in the C++ programming language: The case of novice programmers and expert programmers. *The Journal of Computers in Mathematics and Science Teaching*. 25(2); 197-207.

Wood, D. (1998). *How children think and learn*. Malden, MA: Blackwell.

Zeitz, C.M., & Spoehr, K.T. (1989). Knowledge organization and the acquisition of procedural expertise. *Applied Cognitive Psychology*, 3(4), 313-336.