The Pennsylvania State University

The Graduate School

College of Education

TOWARD A SYSTEM DYNAMICS MODEL OF TEACHING COMPUTER PROGRAMMING VIA DISTANCE EDUCATION

A Thesis in

Adult Education

by

Steven C. Shaffer

Submitted in Partial Fulfillment
of the Requirements
for the Degree of

Doctor of Education
August, 2006

ABSTRACT

This paper explores the use of system dynamics to develop a model of learning to program via distance education. Based on a study which combined a qualitative study of students and instructors in computer programming courses taught through distance education with a meta-ethnographic analysis of the literature of learning to program and distance education, the study identifies and describes 51 variables (17 cognitive, 21 pedagogical and 13 practical) and presents a model which demonstrates how these variables might interact.

TABLE OF CONTENTS

List of figures

List of tables

Acknowledgements

I wish to thank my committee for their help and suggestions during this project, as well as their tutelage during the past several years. I especially want to thank Dr. Melody Thompson for her encouragement and support during the many, many revisions which finally determined what this research was really all about.

In addition, I wish to thank the following people who had a hand in parts of my life which were woven into the accomplishment of this work: Peg Dobrinska, Dennis Dunn, Bonnie Meyer, Karen Norris, Ann Ross, Hoi Suen, and Valerie Thomas.

Dedication

This work is dedicated to my daughter Colleen who has been the light of my life for sixteen years and counting. I truly hope you find the path in life that makes you happy.

I didn't learn much. I just got by. I don't feel at all competent in programming, and I definitely couldn't write a program for a job or other reason.

A programming student

Chapter 1

Introduction

This research project investigates the teaching of computer programming via distance education. Computer programming was originally a skill which was used by scientists, engineers and mathematicians in the course of their work. Later it became a specialized field of study and, although this continues today, once again many non-specialists need to learn how to program as an adjunct to their daily work. Whereas professional programming was one of the fastest-growing job categories throughout the 1980s and 1990s, today job opportunities are not as numerous (U.S. Department of Labor, 2005). In addition, large-scale programming projects are increasingly performed on an off-shore, out-sourced basis (Hatchman, 2004). However, workers who do not identify themselves as "programmers" are increasingly required to do programming tasks for their jobs. By one estimate, the category of non-professional programmers doing programming (typically referred to as *end-user programming*) is 22 times larger than that of professional programmers (Prabhakararao, 2003, p. 281), and many of these people are often doing so in the context of their business-oriented jobs. Thus, teaching programming as a "general education" requirement makes increasing sense. However, introductory programming classes historically have been shown to have very high drop-out rates (Heaney & Daley, 2004; Ramalingam, LaBelle and Wiedenbeck, 2004), possibly due to poor pedagogical practices: "It has long been obvious to educators that students must master the basics before attempting more advanced, abstract endeavors. Somehow, in the fury of technological advancement we have lost sight of this in computer science education" (Buck and Stucki, 2000, p. 75).

Teaching programming differs from other domains because programming comes complete with its own reality check (Buck & Stucki, 2000) in the guise of a program actually executing (or not) on a physical computer. Other disciplines "may be able to fool themselves into believing that their students are exhibiting the higher levels of development" (Buck & Stucki, 2000, p. 76), but this is not a luxury afforded to the teaching of programming. "In short, to write useful software we must deal not only with the computer and its software, but also with the complexities of the natural world in which the application resides" (Jackson, 2003, p. 14).

Failure and drop-out rate is historically very high in programming classes; some possible reasons for this are explored in this paper. Additionally, student achievement is often quite low; for example, McCraken, Almstrum, Diaz, Guzdial, Hagan, Kolikant, Laxer, Thomas, Utting and Wilusz (2001) found, in a multi-national study, that students finishing their first year programming courses scored an average of 22.89 out of 110 points (about 21%) on a standardized assessment of programming ability. And, as Woit and Mason (2003) point out, programming ability often is much poorer than course grades would indicate. Months or even years into programming, many students still maintain conceptual misunderstandings of the field (McGill, and Hobbs, 1996, p. 73).

Historically, research into learning to program has taken two distinct tracks: those who study the nature of computers and computer programs themselves, and those who study computer program*mers,* including how and why they learn to program. The literature available for this second category is significantly more scant than the first. However, since the 1960s there has been a consistent, if small, thread of interest in teaching programming and what is often called *the psychology of programming*. Much of this literature is intertwined with the development of cognitive science, due, perhaps, to the interlacing of cognitive psychology with artificial intelligence and the resulting use of the *brain-as-computer* metaphor which has been a dominant approach in cognitive psychology (Matlin, 1994). Thus, much of the research into computer programmers has been focused either on an implicit efficiency question ("How can we teach someone to be an effective programmer as fast as possible?") or on the cognitive mechanics of teaching someone to program ("What knowledge and task categories exist in the mind of a computer programmer?"). Both of these lines of thought inform this study, though they are not the primary drivers of it. Instead, this research seeks to create a descriptive model combining the task of teaching programming and the task of learning to program, all of which will be referred to here as LP (for *learning programming*).

From a different standpoint, distance education (DE) researchers have attempted to develop models of DE environments using the concept of *systems* – often referred to as systems dynamics or general systems theory (Saba, 1999 and Shaffer, 2005). It seems intuitively obvious that the notion of general systems is connected in some way with computer systems, and thus that there is the possibility of blending these theories into a systems theory of LP in a DE environment (which I will hereafter refer to as LP/DE). However, actual evidence is embryonic in both areas; many system models of DE are based on speculation, and thus a provable *theory* of LP/DE seems far off. But a *model* of LP/DE could be very useful as an organizational device for educational institutions wishing to implement or optimize their offerings in this area.

Research question

This research asks these questions: *What are the elements of an LP/DE program and how do they interact, and how can these elements be combined in a model that illustrates these interactions and provides a guide to practice?*

Initially, as an organizational device, this research ramified LP/DE into the following categories: (1) task analysis; (2) cognitive aspects; (3) socio-economic aspects; (4) pedagogic aspects; (5) assessment; (6) technological factors; and (7) issues specific to the DE part of LP/DE. Task analysis involves delineating the steps involved in programming and learning to program. Cognitive aspects of the learner include such things as knowledge structures, task experience, short term memory limitations and attention. Socio-economic factors include status, educational experience, power relations, social expectations, the economy in general, the demand for programming in the workplace, tuition, and pay scales.

Pedagogic factors include curriculum, group interaction and transactional distance. Assessment, although it could be combined with pedagogy, was separated because it was seen to be a key factor in the success of LP/DE. Technological aids to programming include programming interfaces, simulations, video-on-demand, etc. Issues specific to DE with respect to LP were separated out in order to highlight these factors, making it clear how LP/DE might differ from other LP or DE situations.

Most of these aspects of LP/DE could be (and some have been) the subject of an entire research project – however, the goal of this project is to draw together the existing research in a useful way, specifically to provide programming educators a useful “roadmap” of the various aspects of their profession. Thus, the primary focus of this study is the interplay between the student(s) and the instructor(s), but set into their larger context. The purpose of a roadmap in regular life is to help travelers achieve their objectives by reusing the knowledge of people who have come before.   Such “reuse” in the context of LP/DE draws from the software development field itself, and is exemplified in distance education by the current interest in learning objects (Moore, 2001). The notion is that, although each situation is unique, often some features of a similar situation will allow us to reuse previous work in the current context. This idea is not new, as it is the very basis upon which one might publish a case study. The innovation here is the task of organizing and structuring, using system dynamics, those aspects of LP/DE that are good candidates for reuse.

The research question is purposely circumspect because, as stated above, the state of knowledge of these subjects does not yet warrant a theory, much less specific hypothesis testing. In addition, this research question illustrates the conviction that many, if not all, “theories” of education must be seen instead as potentially useful models, metaphors or narratives to help the professional educator organize his or her work, and not as definitive statements about the nature of educational reality. Education is about relationships, and these are too many or varied to be fully captured in any single unified theory.

Theoretical framework

This research approaches the problem of LP/DE through the lens of systems theory, following Sterman (1994), Saba (1999), and Shaffer (2005). Systems theory is used to attempt to draw a “road map” or big-picture view of the dynamics of the relationships in a LP/DE situation. The research also assumes a cognitive orientation in the lineage of Davies (1991, 1994) and Mayer (1985, 1986, 1988) when looking at the way LP/DE is internally processed by the teacher and the student, and how this interacts with those variables which are *not* internal to these people.

Certainly this project has not arisen *ex nihilo* – the work of Moore and Kearsley (2004) and Saba (1999) presaged it within the field of distance education; the work of Sterman (1994) and Frick (1995) is similar in the area of general education, and Mayer (1985, 1988) developed the beginnings of a systems model of the cognition of programmers during the 1980s.

This approach to studying LP/DE seems necessary at this time in order to develop the foundation for any future work in the field. The domain is too broad and too varied to allow any further advancement without a unifying approach and, given the options, system dynamics seems to be the best approach for developing just such a unified model.

Significance of the study

This research is significant in that it provides guidance for those increasing number of educational institutions that are implementing LP/DE. These programs may be focused on either future professional programmers, end-user programmers, students taking the course as a general elective, or all three. The results could be used by existing traditional institutions, DE-only programs, or for employee training. The model is intended to guide practice; the goal is not to prove that it is true, but instead for the results to be useful. As Toulman (2004) suggests: “we shall do better to develop a new *praxiology* … that asks what procedures are efficacious in any given rational enterprise, on what conditions, and for what practical purposes” (p. 62).

Assumptions and limitations

As stated, the goal of this research was to develop a model of LP/DE which can be used as a roadmap for educators in their development of LP/DE programs, as well as provide the basis for future research. This research was limited by the availability of published resources and the parochial nature of the contexts studied through new empirical data collection during the current study. No claim to universality is made, but there is an anticipation of usefulness to the practitioner and future researchers.

Overview of the paper

The organization of this paper needs some explanation. Since one of the major sources of data is the published works in this field, description of the details of many of these works is presented in the chapter which presents the model. The literature review, in contrast, provides a broad overview of the relevant topics in order to contextualize the decisions made in the study. Some of the topics covered in the literature review reveal themselves as variables in the model, but many are environmental factors in which the model is situated.

The methods chapter describes and justifies the study design, which combines meta-ethnography with a qualitative empirical study. Since meta-ethnography is not well known, some detailed explanation of this method is included in this chapter.

Chapter 4 presents the results of the study, including the primary result of this study: a proposed systems dynamics model of LP/DE which lists, in detail, each variable and describes the proposed relationships between these variables through references to published studies and bolstered by references from new empirical data gathered as a part of this study. In those cases where the conclusion is controversial, I make reference to the potential controversy and explain my reasoning for accepting one conclusion over another.

The final chapter summarizes the findings, lists the limitations of the study, and proposes future research directions.

Chapter 2

Literature Review

This chapter provides important contextual information which underlies and supports the model presented in the results chapter. Since this mixed-methods study takes as part of its data set 80 published articles on learning to program (LP) and learning to program via distance education (LP/DE), the purpose of this review is to examine the literature on the background material underlying this study. The sections of this chapter provide the context within which the model exists, as opposed to the details of the variables within the model, which is left for a later chapter. This literature review is divided into eight sections; each one briefly describes theories that underlie this study.

The first section describes system dynamics and its uses in education theory. System dynamics is the underlying model for this research and thus needs to be explained and justified. The approach of system dynamics is to model the domain being analyzed as multiple interacting variables with (potentially complex) feedback loops. Once feedback loops are incorporated into a system, it becomes impossible to predict its outcomes via the use of simplistic analysis. The claim of system dynamics theory is that without the use of powerful modeling techniques, education theory will forever be trapped within the confines of overly-simplistic explanations for complex phenomena.

Section two discusses the economics of educational decision making. This section explains and justifies the interaction of the wider world with the model presented in the results chapter; this interaction is primarily through the variable in the model called *market forces*. It is possible to incorporate the economic variables discussed in this section into the full model, but this work is left for future development. Instead, this background section describes and contextualizes the possibilities of such extra-model interactions.

The next section briefly discusses the background and context of community colleges in order to contextualize the qualitative data study reported in this paper.

Section four reviews the theories of learning which inform this research. Although drawing heavily on cognitive theory, other approaches such as behavioral, constructivist and action-theoretical are used where appropriate.

The next section presents some background on the task and psychology of programming. It is presented for those readers who may not be familiar with what “goes on” (both externally and internally) when someone is programming.

The sixth section briefly reviews the literature on teaching and learning programming; this differs from the previous section in that this section focuses on the process of *learning* programming, as opposed to the process of *doing* programming. An overview of the types of work done in this area during the past forty years is provided.

Next is an overview of various technologies which impact learning to program via distance education (LP/DE). Both programming and (increasingly) distance leducation are intertwined with computer technology. This section briefly describes some of the technological aspects of the problem; however, this research is not primarily focused on specific computer technologies, and therefore this section is presented as background material to contextualize some of the variables presented in the results section of this paper.

This chapter ends with a few brief concluding remarks about the published sources.

System Dynamics

According to Moore & Kearsley (1996), "It is not possible to improve quality, provide for more students, and lower costs without reorganizing education according to a systems model." With this in mind, the following is a review of literature on the subject of the use of systems concepts in education, starting with a brief review of a few key background references.

Systems thinking

Systems theory, system dynamics, and similar phrases are often found in the literature of education (in general) and online education (in particular) (Shaffer, 2004 and Shaffer, 2005). Sterman (1994) offers an excellent, detailed introduction to the use of systems theory in education. Two main strands of systems thought exist: systems theory and system dynamics. Systems theory was initially developed by biologist Ludwig von Bertalanffy as a rigorous method of describing the structure and mechanisms of organic systems. He was very concerned about "the danger that general system theory may end up in meaningless analogies” (p. 35) and was at pains to point out that “general system theory is not a search for vague and superficial analogies” (von Bertalanffy, 1968, p. 35). System dynamics (Forrester, 1968), the approach taken in this paper, comes out of the tradition of cybernetics research and is primarily interested in the dynamics of feedback loops which, it is claimed, lie at the core of all real-world phenomena.

Systems thought in an educational context is problematic; authors sometimes write about looking at an educational situation from “a systems perspective” but then do not really apply the tools and techniques of systems theory or system dynamics.

Dynamic systems

A dynamic system is one which inherently incorporates feedback loops. These loops are of two broad kinds: positive (or self-reinforcing) or negative (damping). An extremely common example of a positive feedback loop is any exponential growth situation – for example, the growth of a rabbit population in the absence of constraints (predators, food shortages, etc.) The reason that this is a positive feedback loop is that, starting from some small number of rabbits, each generation is built upon the previous (see figure 2-1). An example of a negative feedback loop is a boat pulling into a dock – while still a half a mile away, the pilot will make large vector adjustments to steer the boat toward the destination; as the boat approaches the dock, navigational changes will become smaller and smaller (see figure 2-2). Negative feedback loops exhibit the quality of damping (decreasing the amplitude of an oscillating system) which is characteristic of goal-driven systems.

Each of these types of system change can be described by the following causal loop diagrams. Figure 2-3 shows a *positive reinforcement loop*, indicating the number of bunnies increasing as the number of newborn bunnies increases and vice-versa. Figure 2-4 shows the causal loop involved in docking a boat; as the pilot approaches the dock, the difference between the current position and the goal decreases; additionally, the amount of correction that the pilot needs to make also becomes smaller.

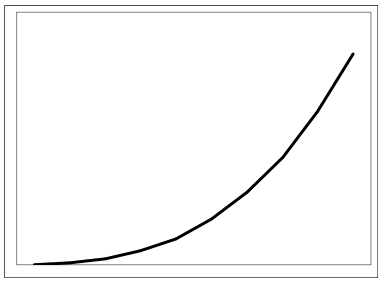

Figure 2-1
Exponential growth

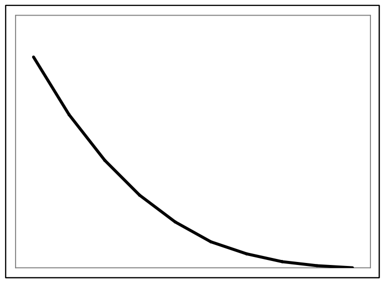

Figure 2-2
Exponential dampening

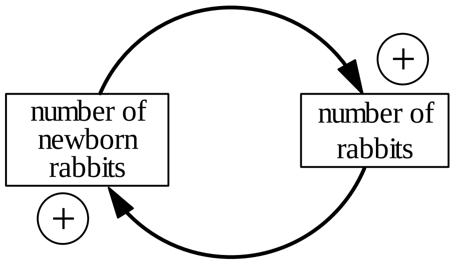

Figure 2-3
Positive reinforcement loop

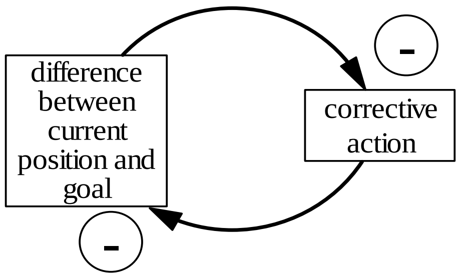

Figure 2-4
Negative reinforcement loop

Real life is filled with such systems; the problem is that people do not make good inferences about even the simplest systems, even if given full information. Researchers have demonstrated this (Sterman, 2000 references specific studies), to the point that is has become a staple of organizational management seminars to demonstrate these results using simulation systems.

Adding to the complexity of systems is the human-in-the-loop, where the human his- or herself is learning and thus changing the system dynamics. Most people operate in the single loop learning mode (figure 2-5) – in this situation, the person adjusts his/her decisions based on the information feedback from the "real world." Unfortunately, the human often does not update his/her mental models and/or strategies. This can present itself as an obstinate "damn the torpedoes; full speed ahead" attitude. A more complex adaptive behavior is to adjust one’s mental models and strategies as one processes feedback from experience. Thus, one might start a project with a simplistic view of the situation, but adjust one's strategy as the complexity of the situation is realized. This is referred to as double loop learning (figure 2-6).

Argyris and Schon (1974), who brought the notion of double-loop learning to prominence, discuss the interaction of single- and double-loop learning: “Double-loop learning does not supercede single-loop learning. Single-loop learning enables us to avoid continuing investment in the highly predictable activities that make up the bulk of our lives; but the theory-builder becomes a prisoner of his programs if he allows them to continue unexamined indefinitely. Double-loop learning changes the governing variables (the ‘settings’) of one’s programs and causes ripples of change to fan out over one’s whole system of theories-in-use” (Argyris and Schon, 1974, p. 19).

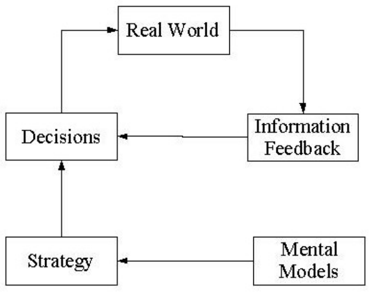

Figure 2-5 – Single loop learning

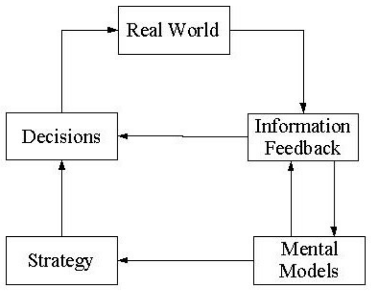

Figure 2-6 – Double loop learning

An important confounding aspect of real-world systems is delay. Delay is inherent in receiving feedback and, the longer that delay is, the harder it is to operate effectively in the system. In an educational context, this could manifest itself in wide swings in policy between, for example, literacy and math skills, as follows: (1) at some point, a problem with literacy is noticed; (2) a correction is undertaken; (3) without having allowed enough time for the correction to be effective, it is noted that the problem still exists; (4) more corrections are undertaken; (5) as the earlier corrections take hold, an unintended consequence of negative effects on math skills is noted; (6) a correction for the math skills component is undertaken; (7) meanwhile, the effects of the second literacy correction begin to develop, causing (8) further deterioration of math skills, which causes (9) further corrections to the math issue, which has already begun to negatively effect the literacy component from the first adjustment, so (10) a decrease in literacy is noticed and the cycle begins again at step (1). The inability of people to wait long enough to see the results of their machinations is the cause of many strange human behaviors, such as real estate boom/bust cycles, stock market fluctuations, and aircraft pilot over-correction and oscillation.

This example assumes that education is a zero-sum game, meaning that any gain in one area means a corresponding loss in another. While this is not *exactly* the case, in the context of limited educational resources it can be seen to be. (If, for example, a policy included adding thirty more days to the school year, then it would cease being a zero-sum game until that extra time was thoroughly used.)

Feedback is pervasive in daily life and is an essential ingredient for learning, as described for example in Vygotsky (1980, 1986). The problem is that most people are unable to process the entire complexity of even the simplest system. “The complexity of our mental models vastly exceeds our capacity to understand their implications” (Sterman, 2000, p. 37). Most people do not employ sound logic or scientific reasoning even in simple tasks; in a recent informal study that I performed, only 10% of junior and senior computer science and engineering students (n=55) got the right answer on a logic problem with only four variables; this is somewhat better than the general population result of only 4% for the same problem (Sterman, 2000).

The best use of system models is in situations where controlled experiments are economically unfeasible, unethical or impossible. The only way to work within complex contexts is to use what are often called “micro-worlds” – simulations, models, etc. – which allow us to “play” with the variables to see the results of various situations. Scientists and engineers do this all of the time, as do airline pilots and architects. Simulations allow us to begin to comprehend the complex forces under study in a way which we would not be able to without them.

However, there is an important difference between theory and experiment. Theory helps focus experiments and draw the results of those experiments together. This is where simulations and models can be used, but they are not an alternative to experimentation, any more than piloting a 757 flight simulator is the same as actually flying the plane itself. At some point theoretical models must give way to the world of experimental evidence; it is only by doing so that we will be forced to model those messy aspects of the real-world situation that might not easily fit an elegant model.

Social systems dynamics models

Three aspects of social systems exist which make them inherently “loopy”: (1) policy resistance, (2) the principle of bounded rationality, and (3) the law of unintended consequences (Sterman, 2000). Each of these is briefly described below.

Policy resistance is the aspect of social systems which happens when agents withhold participation in an established order because it is perceived to not be in their own best interests. A well-quoted reference to this phenomenon is President Truman's statement regarding incoming president Eisenhower: “He’ll sit here, and he’ll say, ‘Do this! Do that!’ And nothing will happen*.* Poor Ike—it won’t be a bit like the Army. He’ll find it very frustrating” (Neustadt, 1960).

The *principle of bounded rationality*, for which Herbert Simon won the Nobel prize in economics, states that the capacity of the human mind for handling complex problems is much smaller than the size of “real world” problems and thus the standard economic assumption of a “rational decision maker” is flawed. He replaced the standard model with one which he called *satisficing*, the practice of selecting and accepting a goal or solution that is *good enough* within the confines of the limited search time and inadequate information (Hellriegel, Slocum and Woodman,1995, p. 622).

The *law of unintended consequences* is the notion that, in the absence of full knowledge of the situation (i.e., always), decisions will have effects which were not anticipated. An example in an educational context might be the rapid increase in enrollment in a course due to the unintended consequences of a change to the curriculum that accidentally allows students to fulfill two requirements by taking a single course.

Creating non-simplistic system models of social systems can help to ameliorate these types of problems by giving the analyst the tools needed to grasp the outcomes of policy decisions in complex situations. The challenge is to move from generalizations to specific actionable theories with high-probability predictable outcomes. It's important to note that one can model only problems, not entire systems, because you must make decisions about what aspects of the problem are salient. Full models of complex systems would be as complex as the system itself and therefore of no use. This is not to say that significant portions of existing models can not be “borrowed” within other contexts, just that it is not possible to create a “one size fits all” model of any complex social system.

The nature of systems

Strauss (2002) analyzes system theory from the standpoint of several perennial philosophic questions: (1) the one versus the many (atomism versus holism); (2) the whole-parts relation; (3) constancy versus change; and (4) *entelechy* or vital force. Of particular interest for this research is the question of the ontological status of a system, that is: does a system exist, or do only the parts of the system exist? For example, a university is made up of people and buildings, but does the university itself exist as an entity? If so, in what way? From a philosophic standpoint, Strauss seeks to determine if systems as diverse as universities, calculus, and pork-belly markets have an independent existence, and what exactly the nature of that existence is.

Saba (1999) discusses the difference between a systems view and what he refers to as the “physical science” view of education, specifically in the context of distance education. After briefly discussing some of the history of this approach, Saba then presents some methodological concerns. Chief among these is the limiting nature of the reductionist approach (which he claims is inherent in the physical science view). Although he does not use this term, he in effect charges that the physical-science approach to educational research lacks ecological validity because the tasks that subjects are asked to perform are decontextualized. “The organism,” he states, “is treated like a machine whose task is to associate inputs and outputs” (Saba, 1999, p. 27). Another problem with reductionism, Saba points out, is that “the imperative of time in all aspects of human existence, including learning, is rarely brought into the picture” (p. 28). Here Saba seems to be referring to that aspect of learning which happens inside of a delay loop; this type of phenomena is better modeled using a system dynamics approach. Saba (2003) sees systems theory as “the quintessential (pragmatic) tool for understanding relationships between things and not looking for a single answer to a problem within the confines of a dogma” (p. 11).

Smith and Dillon (1999), responding to Saba (1999), defend what Saba refers to as the “physical science” approach to educational research. Their main defense is that such research is necessary, though not sufficient, in order to develop an overall model of distance education: “Thus, we argue that distance education must be examined as a system, but to do so requires looking at the system and the variables that make up the system, sometimes a few at a time” (Smith and Dillon, 1999, p. 34). Overall, the authors make a good point: One can not have an overarching theory without some data from which to construct it; however, the main point of Saba (1999), though he does not state this explicitly, seems to be that much research is published as disembodied pieces of information that lack any coherence.

Systems theory and software simulation in education

One of the chief values of a systems theoretic account of anything is that a well-defined system model lends itself to software simulation. It is critical to recognize that “regardless of the form of the model or the technique used, the results of the elicitation and mapping process is never more than a set of causal attributions, initial hypotheses about the structure of a system, which must then be tested. Simulation is the only practical way to test these models. The complexity of the [interactions] vastly exceeds our capacity to understand the implications” (Sterman, 1994, p. 321).

Frick (1995) attempts to develop a systems theory of education (in general, not just distance education), drawing from the work of Maccia and Maccia (1966), who propose 201 hypotheses on the relationships among properties of educational systems. Frick proposes to build software simulations of the complex interrelations within an educational system, and to use these simulations to help educators and administrators to propose innovations within their systems. Using such a modeling approach, King and Frick (1999) demonstrate that the results of the educational situations they were studying could have been predicted from their models.

Saba and Shearer (1994) note that “research in distance education has been primarily program based” (p. 36) and that the few empirical studies have measured factors such as stakeholder satisfaction. They propose that in order for the field of distance education to mature, there is a need for empirical studies to verify the conceptual foundations of the field (which is surprisingly similar to the argument in Smith and Dillon, 1999, which disputes Saba, 1999). Thus a symbiotic relationship exists between variables and system simulations; one needs to know what the variables are in order to create the simulation, but also one can not know what the cogent variables are until one begins to develop the system. The solution to this predicament is to take an educated guess at the variables at first, then to test and hone the relationships as well as possible.

Anderson (2004a) believes we are still early in the development of a theory of distance education. He believes that any full theory would be premature at this point; however, “a first step in theory building often consists of the construction of a model in which the major variables are displayed and the relationships among the variables are schematized” (Anderson, 2004a, p. 48). Anderson offers such a model, combining attributes of learning with the “affordances” of the Net, together with a theory of the role of interaction in online learning. Building from the work of Moore (e.g., Moore and Kearsley, 1996), Anderson elaborates a model of the interaction between the three “agents” in educational interactions: teacher, learner and content, creating six possible interactions (learner-teacher, teacher-content, etc.). Next, the author describes a more complex model which includes the “knowledge/ content interface” and aspects of communication and outside support. Generally, although the author does not use the phrase system theory, his model might be a good basis on which to build a system theory of online learning. The author concludes that his model “illustrates most of the key variables that interact to create online educational experiences and contexts” (p. 55).

At their base, educational variables are many and complex, and attempts to restrain them into a hierarchy is only for our own ease of use and is eventually doomed to a false simplicity. Real-world systems are of a complexity that simple deterministic models (which were all that we could create before the advent of the computer) are doomed to fail. An obvious example of this is weather forecasting, which has made enormous strides during the last twenty-five years due in a large part to the use of computer models (Ostby, 1999).

Smith and Dillon (1999) claim that “predictive theory in distance education is premature and, perhaps, in the strictest sense, unattainable” (p. 35). However, system dynamics has uses other than prediction; instead it can be used to nurture a better understanding of the interactions of the variables in a system in order to avoid overly simplistic treatments for complex problems. As Sterman (1994) declares, “the challenge facing all is how to move from generalizations about accelerated learning and systems thinking to tools and processes that help us understand complexity, design better operating policies, and guide organization- and society-wide learning."

Developing a system dynamics model of DE

Drawing from the research, figure 2-7 is a list of *some* of the many variables which are thought to effect educational systems. There is a tendency by researchers, when faced with a large list of variables, to attempt to ramify the list into categories – this is a natural and useful tendency when dealing with, for example, a grocery shopping list. Thus categorized, one can use the list as a reference while in each of the areas in the grocery store (produce, frozen food, etc.).

content experts	type of material (knowledge-based, task-based, etc.)
other sources of knowledge	feedback
student characteristics	learner support
instructor experience	preparation time
administration	synchronous/asynchronous
student access to resources	cost of developing the materials
course material	pacing of the course
assessment of student's progress	learner-content interaction
learner achievement	learner-instructor interaction
learner attitudes	learner-learner interaction
communications media	student's ability with the technology
technologies	student anxiety
course design	student characteristics
cost-effectiveness	student motivation
class size	policy resistance
length/schedule of class	full vs. part-time instructors
reasons for taking the course	inter-student competition
prior educational background	inter-departmental competition
scheduling	inter-organization competition
accreditation	

Figure 2-7: Some variables in online education systems

The trouble with this approach is that educational variables are interconnected, unlike most grocery items. The concept of interconnected grocery items would be typified by the following rule: *Buy frozen corn only if there is no fresh corn in the produce section.* This is especially complicated if the frozen food section comes before the produce section in your tour around the store – during your first visit to the frozen food section, you do not know whether or not to pick up the frozen corn. In this case you have two possibilities: pick up the frozen corn and perhaps have to loop back to put it back, or do not pick up the frozen corn and risk having to loop back and pick it up if there is none available in the produce section. This is an example of a simple coupling between variables that would be hard to model if the list was broadly categorized as produce, frozen food, etc. Thus, the very tendency to use categorization to simplify a problem actually restricts the quality of the model. The tools of system dynamics provide the ability to handle the complexity of a problem without having to resort to power-reducing heuristics.

As previously mentioned, modeling educational environments via system dynamics is not unheard of. As mention previously, Frick (1995) proposes to build software simulations of educational systems (see also King & Frick, 1999). If even mildly successful, such a tool would allow educational theorists and policy-makers to test out the results of their theories in a way that is (a) deeper than can be done via an “armchair” analysis alone and (b) safer than testing theories on humans. Such an approach has been taken in the pharmaceutical industry for years; *it can never replace actual implementation with humans*, but it can at least help the researcher to focus on those ideas which have a higher potential for success. Frick proposes that the reason this approach has not been developed in education thus far is that most people do not think in a systems manner; he contends that a major shift in thinking must occur before an approach such as his will gain general acceptance.

The limits and usefulness of models

System dynamics modeling has its limitations:

> "Simplification and abstraction, in the form of a system model, inherently involves a process of selection by the researcher. It is generally impossible to include every possible relevant factor in the model. Alternatively, over-simplification risks irrelevancy. The challenge is to find the right balance between *simplified* (for easier digestion and analysis) and *representation of reality* (for applicability and usefulness)" (Johnstone and Tate, 2004).

Any model is an approximation; the art of modeling is incorporating *all* and *only* those things which are germane to the situation at hand (Sterman, 2000). For this reason, system dynamics models can only be developed for specific situations; they do not allow for the creation of a "unified theory of everything." Full models of complex systems would necessarily end up being as complex as the system itself. However, as models are developed, it is possible to re-use certain aspects of a model for similar situations, thus giving the researcher a starting point from which to develop a model of a particular situation. Such a model can only be relied on if it is validated: “Regardless of the form of the model or the technique used, the results of the elicitation and mapping process is never more than a set of causal attributions, initial hypotheses about the structure of a system, which must then be tested. Simulation is the only practical way to test these models” (Sterman, 1994, p. 321).

In addition to studying the organizational aspects of education, system dynamics has the potential to develop models of the “human in the loop” (Harris, Iavecchia, Ross, and Shaffer, 1987), thus enabling educational researchers to analyze the effects of pedagogical decisions and course design. By drawing together research from a number of disciplines such as education psychology and cognitive psychology, it should be possible to model the overall success rate of various course elements given a statistical distribution of individual learner differences. It is important to note that this is *not* the same thing as reporting an average effect size, etc.; the statistical aspect is limited to the input aspects of the model in the form of demographic and cognitive differences of the students, but each output result is individualized to the statistically generated individual. This is analogous to the difference between the square of sums and the sum of squares: the focus here would be on statistically generating the example student and then identifying the result that student would achieve based on the model. What we could learn from such a model is the impact of individual student differences on policy decisions and vice-versa.

Using system dynamics to describe distance education

Distance education can be well modeled via system dynamics, specifically due to system dynamics' ability to capture the “small-changes-cause-large-effects” nature of systems. If one breaks an educational system into, say, five easy pieces, that system will not be able to exhibit complex behavior and thus any modeling of distance education will stay encapsulated within its hierarchical category and not effect the system as a whole. By separately modeling dozens or hundreds of interacting variables within a system, it is possible to witness *emergent behavior*. The notion of emergent behavior is central to current systems thinking (Crutchfield and Mitchell, 1994) and can only happen within the context of a system of sufficient complexity. How a small factor can grow into a major changing force in an existing environment can be demonstrated by the Tacoma Narrows Bridge collapse in 1940 (see http://abel.math.harvard.edu/archive/21b_fall_03/tacoma/ for a video clip) where a relatively small force of wind (35 - 46 mph) at the resonance frequency of the bridge caused the bridge to sway wildly and eventually collapse. At the time the bridge was built, engineers did not have a strong notion of all of the forces at work on a suspension bridge; after the collapse, they learned more. Updating models based on experience is the central activity of model building, whether the domain is bridges or LP/DE.

The economics of educational decision-making

This section delineates a set of assumptions which underlie the current work. The model presented in chapter 4 of this paper must be seen within the larger context of the economics of educational decision making. Although this review draws heavily from Arai (1998), the organization and presentation are my own.

The *net present value/internal rate of return* method is the notion that students look at attending higher education as an economic function similar to a capital investment. In this model, a student (probably unconsciously) calculates that if s/he attends college (thus postponing income production for approximately 4 years) that s/he will recoup this cost by virtue of later earning power. Calculated into this model are the following variables:

- cost of the education
- cost of lost income
- expected increase in income after college degree is obtained
- current interest/investment rates

Based on this model, the decision to attend college is a strictly rational decision. However, capital market imperfections cause some perturbations in the model. For example, the student may not be able to raise the capital to pay for the education even if it would be advantageous in the long term. This means that access to higher education would be skewed toward those whose families which could afford it. These factors can be ameliorated through loose borrowing practices (e.g., student loans), lowered tuition rates, or scholarships.

Of course, students do not go to college just because of a cost/benefit analysis: “The real benefits of higher education, however, should include the psychological satisfaction which can be gained from jobs college graduates take. Suppose an individual gains great psychological satisfaction from teaching high-school students. This can not be gained without going to college and obtaining the necessary qualifications. Thus, this satisfaction is unambiguously the benefit of higher education, but it is not measured in the above methods.” (Arai, 1998, p. 27). Arai goes on to note the advantages of working conditions, fringe benefits, etc., which are often associated with jobs that require college degrees as opposed to those that do not.

Arai analyzes the possible causes of the economic value of higher education. It is possible, he proposes, that the reason people with college degrees receive more pay is not due specifically to their degrees but instead to their intelligence, which is tightly correlated with one's ability to attend and graduate from college. He refers to research that statistically removed the native intelligence component from the data and found that about two-thirds of the increased income is due specifically to the college degree.

An economic phenomenon known as a “cobweb cycle” (Arai, 1998, p. 31) causes some complexity in the economic forecasting of the value of a college degree; for example, US college enrollments are counter-cyclical (Dellas and Sakellaris, 2003). The complex loop-back relationship makes it very difficult to predict certain economic aspects of attending post-secondary training.

Models of college attendance which are related to human capital theory suffer from some problems “in the sense that it does not necessarily have empirical support” (Arai, 1998. p. 32). Human capital theory assumes the intermediary process of *increase in productivity*, which is not borne out by the data.

An alternative way of looking at the issue is to consider the economic value of a worker as a combination of higher education and post-education training. The later might take the form of on-the-job training, off-the-job training (e.g., continuing education), and job experience. In this model, employers invest the cost of further education and training of their employees in order to increase worker productivity. This differs from the above theory due to the fact that it involves investment in training for employees already on the job. In the model, the main difference between the curves describing employer investment and employee earnings is the curve for the college graduates is offset higher, indicating that although the growth rate is similar, the college graduate starts higher and stays consistently higher throughout his/her career.

The previous models are described as a precursor to the notion of college-going behavior as *signaling*. Arai (1998) describes this as follows:

> “There is a theory of college-going decision making which is quite different from human capital theory. This alternative theory is based on the idea that the role of the college degree is not to certify the knowledge and skills acquired in college, but simply to convey information to society about the degree holder's productivity. The

> productivity may be innate, or may have been acquired (at home) by the time of the start of college education. According to this theory, those who have high productive capabilities go to college so that firms can identify degree holders (or highly educated people) as more productive and pay them more. Here, college education does not enhance the productivity of the students, but only plays a role of judging how productive each individual originally is... . This 'signaling' theory is very powerful and competes with human capital theory to explain college-going behavior" (Arai, 1998, p. 47).

It may be that people attend college in order to inform society that they are high-performing. Employers wish to know a potential employee's abilities, which they can not know directly and instead must infer from readily available data. So, an employer might decide to hire a college graduate over a non college graduate based on the notion that the former is higher-performing. This is different from the human capital model in that there is no assumption that college has made the person more productive, just that only more productive people graduate from college. This means that a student may still choose to go to college knowing that it will not improve his/her productivity at all.

However, there is more than a financial cost to attending college: there is a cognitive cost. A student might be able to graduate from college with little cognitive effort, or it might take a great deal of effort. For some, the loss of leisure time might be a significant cost. Indeed, different colleges, and even different majors, carry a variety of signals: colleges and majors which are thought to be more difficult carry a stronger signal of ability than those which are not perceived to be as difficult. Majors which are perceive to be “serious” indicate a lack of frivolity to some employers while perhaps signaling a lack of creativity to others.

Employers implicitly indicate that they will pay more for a college graduate with certain signaled attributes; thus, each graduating high-school senior will make his educational choices based what s/he knows about this implicit agreement, plus the calculated economic and cognitive costs of the various options.

However, “the basic reason that (this) model cannot convince ... people is that it assumes those with the same productive capabilities or intelligence face the same signaling costs. There are, however, many cases in which this assumption does not hold” (Arai, 1998, p. 55). This brings us back to capital market imperfections; something as simple as living near a college can alter the signaling cost for a potential student. Economic class can make the financial cost to some prohibitive while others are able to afford college without hardship. Not only the cost of the education itself, but the cost of the preparation for college must be factored in: the cost of living in an area with good schools, the cost of private school, and the affordability of prep classes are some obvious examples.

Sometimes an employer is looking for traits which can not be learned in college, such as what might be called “polish.” In such a case, the employer might implicitly be looking for an employee from a wealthy family, under the assumption that the employee will have this trait. Thus, the employer might hire a graduate from a prestigious and expensive institution from a less challenging major instead of one from a technically-oriented school with a more difficult major course of study:

> "In the Spence model students need to study hard to obtain degrees, since if everyone could obtain a degree without studying hard, higher education could not distinguish the productive from the unproductive. In contrast, when higher education plays the role of signaling wealth, it need not require students to study hard since what is more important to them is the payment of pecuniary costs. Still, they may need to pretend to be learning advanced valuable knowledge in order to avoid antipathy toward them. Studying is also important to the extent that it prevents intelligent but poor students from working part-time to pay for tuition, fees and costs of living ... As an economy approaches maturity, the above tendencies are reduced and family income and wealth will become less important in determining an individual's signaling costs. What becomes relatively more important is the mental and time cost of acquiring a certain amount of knowledge" (Arai, 1998, p. 62-63).

Arai's analysis models what is happening with distance education; as our economy matures, the issues of economic class become less important while the cognitive and time costs of education ascend to prominence. Of course, there is still a social value placed on a degree from Harvard that is not attributed to one from most other universities, but as the “slush” in the economy is removed (for reasons of, among other things, globalization), employers are less interested in such amorphous qualities as “polish” and are more interested in results which can be calculated economically.

The context of community colleges

Because some of the empirical data for this study was obtained from community college students and instructors, this section briefly outlines the context of the community college in the United States. Although community colleges originally focused on student transfers to four-year institutions, over the past twenty years their mission has increasingly been to offer terminal associates degrees and continuing and vocational education to adults (Kane & Rouse, 1999). Currently the average age of students enrolled in community colleges is 29 (American Association of Community Colleges, 2005) and about one-third of all high school graduates will attend a community college some point in their lives, making these institutions important providers of adult education.

Tuition is often significantly lower than other higher education institutions, owing to the financial support of state and local governments. The majority of instructors at community colleges have a master's degree as their highest degree, whereas in four-year colleges, most instructors have a doctorate (Kane and Rouse, 1999). Community college faculty spend more time teaching than faculty at four-year colleges (Kane and Rouse, 1999).

Students often attend community colleges due to lower costs and/or because they need remediation before being ready to attend a four-year institution (Kane and Rouse, 1999). Community colleges are “uniquely positioned to contribute to state and local workforce development” (Office of vocational and adult education, 2005). By one estimate, “each year of credit earned at a community college is associated with a 5-8 percent increase in annual earnings – which happens to be the same as the estimated value of a year's worth of credit at a four-year college” (Kane & Rouse, 1999, p. 73). Students who enroll in a community college after being laid off of their jobs obtain an economic benefit over those who do not, but still do not usually come back up to the level of pay they had previously earned (Kane & Rouse, 1999, p. 73). One economic analysis demonstrated that “a year of community college increases earnings by an amount roughly equal to the value of the resources used to produce that year” (Kane & Rouse, 1999, p. 79), which may account for the interest shown by community colleges in cutting costs, and may help account for the fact that they were quicker to adopt distance education than either public or private four-year institutions (Tabs, 2003).

In summary, community colleges are on the forefront of adult, continuing and higher education, are often the institution of first resort for vocation-oriented training, and are especially amenable to teaching via distance education. All of these attributes make community colleges a viable choice of institution for this study.

Theories of adult learning

This study focuses on post-secondary learners, and thus it is important to have a theory of how learning happens in this target population. This section presents an overview of various theories of adult learning, with particular emphasis on language learning, since learning a programming language is seen to be a primary component of learning to program.

Vygotsky

The writings of L. S. Vygotsky have been adopted by many adult education theorists as the genesis of much of current critical theory and, especially, activity theory. This section reviews Vygotsky's (1978) views with specific reference to the subject of language and learning. Vygotsky's purpose is to “characterize the uniquely human aspects of behavior, and to offer hypotheses about the way these traits have been formed in the course of human history and the way they develop over an individual's lifetime” (p. 19). He sees the essence of complex adult human behavior in the dialectical unity of practical intelligence and sign use (p. 24); in fact, he says that “the most significant moment in the course of intellectual development, which gives birth to the purely human forms of practical and abstract intelligence, occurs when speech and practical activity, two previously completely independent lines of development, converge” (p. 24).

Through his observation of children, he noticed that tool use starts to become more complex (less “ape-like”) with the advent of language. Initially, children will speak out loud while attempting complex tasks; later, he believes, the verbalization is internalized.   Through this speech, children “acquire the capacity to be both the subjects and objects of their own behavior” (p. 26): “The child begins to perceive the world not only through his eyes but also through his speech. As a result, the immediacy of 'natural' perception is supplanted by a complex mediated process; as such, speech becomes an essential part of the child's cognitive development” (p. 32) resulting in “the inevitable interdependence between human thought and language” (p. 33) wherein “the system of signs restructures the whole psychological process” (p. 35).

Vygotsky specifically rejects the view that the child suddenly and irrevocably deduces the “relation between the sign and the method for using it. Nor does she intuitively develop an abstract attitude derived, so to speak, from the 'depths of the child's own mind.' This metaphysical view, according to which inherent psychological schemata exist prior to any experience, leads inevitably to an *a priori* conception of higher psychological functions” (p. 45). This can be seen as a dismissal of the Chomskyian view of language acquisition (recognizing that Vygotsky died before Chomsky came to prominence). There is the possibility that Vygotsky and Chomsky are speaking at two different levels of abstraction; Vygotsky is perhaps rejecting the notion of natural categories, while Chomsky is referring to mental “hardware” which accepts certain sequences as grammatical while rejecting others as ungrammatical. However, Vygotsky clearly accepts the notion that there are “elementary processes, of biological origin, on the one hand,

and the higher psychological functions, of sociocultural origin, on the other" (p. 46).

All of this has interest to the current study in that it highlights (but does not resolve) the issues of (a) computer programming as a social construction and (b) the relative "naturalness" or "unnaturalness" of the grammar of programming languages. Further, Vygotsky distinguishes between tools and signs in that tools are externally oriented, whereas the sign is internally oriented; it "is a means of internal activity aimed at mastering oneself" (p. 55). The interplay between tools and signs in computer programming is a fascinating area for further study.

No discussion of Vygotsky's work is complete without a reference to his concept of *the zone of proximal development*, which is "the distance between the actual development level as determined by independent problem solving and the level of potential development as determined through problem solving under adult guidance or in collaboration with more capable peers" (p. 86). Although Vygotsky's work concentrated on children, there is no reason not to believe that this same zone exists in adults; if so, finding this zone is the job of the instructor. The zone of proximal development is a fundamental notion which is built into the model developed in this paper through the use of what has come to be known as *scaffolding*.

Social constructivism

The notion that reality is socially constructed is commonplace in adult education theory; one of the best known treatises on this subject is Berger and Luckmann's (1967) book *The Social Construction of Reality*. The following summary of social constructionism is taken from that work. Berger and Luckmann's main argument is that “no human thought (with only the afore-mentioned exceptions) is immune to the ideologizing influences of its social context” (p. 9). These afore-mentioned exceptions are mathematics and some parts of the natural sciences. As opposed to some later writers on social constructionism (see, for example, Watzlawick, 1984), Berger and Luckmann make a point of defining reality, which they say is “a quality appertaining to phenomena that we recognize as having a being independent of our own volition (we can not 'wish them away')” (p. 1). They posit that people are “conscious of the world as consisting of multiple realities” (p. 21) but that there is one paramount reality – the reality of everyday life which “is organized around the 'here' of my body and the 'now' of my present” (p. 22). Our common language objectifies as it is employed to represent novel circumstances; one “distorts” the reality of a situation as soon as it is encoded in the common language.

Berger and Luckmann (1967) believe language to be the “most important sign system of human society” (p. 37), although it is not the only one. Language is most important because of the “immense variety and complexity” (p. 37) of the ideas that it can symbolize. It is the ability that language gives us to detach from the face-to-face situation that gives it power: “The detachment of language lies much more basically in its capacity to communicate meanings that are not direct expressions of subjectivity 'here and now'" (p. 37). Language gives we humans the ability to speak of things in the past, in the future, and even of things which do not exist at all.

This is an important aspect of the abstract nature of programming, which is fundamentally a way of abstractly expressing a series of actions in a future which may never occur (Blackwell, 2002). Berger and Luckmann's theory has interest for the study of learning to program because, as pointed out by Buck & Stucki (2000), programming comes complete with its own "reality check," manifested in a program that works (or doesn't). Therefore, the student must create an interpretation of the reality of the programming problem into an abstract representation that drives the creation of the computer program, which in turn is used to verify that the abstract representation is accurate. But this translation inserts bias since it requires that the programmer differentiate what is important from what is not.

Berger and Luckmann (1967) see language as a crystallizing and stabilizing force that forces us into its patterns (p. 38). Language abstracts our experiences: "As it typifies, it also anonymizes experiences, for the typified experience can, in principle, be duplicated by anyone falling into the category in question ... In this way, my biographical experiences are ongoingly subsumed under general orders of meaning that are both objectively and subjectively real... Language now constructs immense edifices of symbolic representations that appear to tower over the reality of everyday life" (p. 39-40). An example of such an immense edifice in the context of this paper is the World Wide Web, which structures and drives much of “reality” in modern life. Berger and Luckmann state that “language provides the fundamental superimposition of logic on the objectivated social world” (p. 64). At first, the only language of the web was HTML; it quickly became apparent, though, that this language was not sufficiently powerful and thus other web-based languages were born.

Berger and Luckmann also discuss the automaticity of certain types of pragmatic knowledge that they refer to as *recipe* knowledge. This is the kind of information which is compiled into automatically executing chunks and not ordinarily considered once the activity is learned; for example, driving a car. Having this behavior habitualized gives us the advantage of economy of effort; it lessens one's cognitive load, although they do not use the term: “By providing a stable background in which human activity may proceed with a minimum of decision-making activity most of the time, it frees energy for such decisions as may be necessary on certain occasions” (p. 53). This concept is central to several theories of learning to program.

Transformative Learning

An approach to adult education known as transformative learning is primarily associated with Jack Mezirow, and the following summary is based on his Transformative Dimensions of Adult Learning (1991). Mezirow begins by defining the goal of his theory, which is “to elucidate universal conditions and rules that are implicit in linguistic competence or human development. Specifically, it seeks to explain the way adult learning is structured and to determine by what processes the frames of reference through which we view and interpret our experience (meaning perspectives) are changed or transformed" (p. xiii).

According to Mezirow, learning is bound together with socialization, the nature of which is inherent inequality (p. 2). Socialized thinking processes create the "boundary structures for perceiving and comprehending new data" (p. 5). People who wish to develop more control over their lives need to overcome the cognitive constraints of their socialization. He believes that "in order to be free we must be able to 'name' our reality, to know it divorced from what has been taken for granted, to speak with our own voice" (p. 3). With respect to the current research, learning to program can be seen to be a socialization into a group but also as a freeing experience in that one is thereafter free to reject the computer-based worldview.

In contrast to some other theorists, Mezirow specifically notes that meaning is construed prelinguistically as well as linguistically (p. 4). Mezirow differentiates between *construal* and *comprehension*; construal "occurs outside of our local awareness and without the use of language categories" (p. 12), whereas "comprehension is a process of making an experience coherent by using categories acquired through language" (p. 12). This might manifest itself in the sense of "flow" experienced by some programmers (Pilke, 2004). Mezirow believes that intuition, that is “the ability to have immediate, direct knowledge without the use of language or reason” (p. 14), plays a key role in adult learning. Studies show this to be an attribute of expert programmers (Arzarello, Chiappini, Lemut, Marara, and Pellery, 1993).

Andragogy

The concept of andragogy is associated principally with the work of Knowles (e.g., Knowles, 1989), and includes the assumption that an adult learner is someone who brings life experience and self-knowledge to the table in a learning environment. This difference from teaching children has several consequences, including “the great emphasis …placed in adult education on individualization of learning and teaching strategies … (and) techniques that tap into the experience of learners, such as group discussion, simulation exercises, problem-solving activities, case method, and laboratory methods” (Knowles, 1989, p. 83-84).

However, the life experiences that adults bring to learning can also have a detrimental effect, in the guise of “mental habits, biases, and presuppositions that may cause us to close our minds to new ideas, fresh perspectives, and alternative ways of thinking” (Knowles, 1989, p. 84). Within the context of computer technology, students might have a problem breaking through their habitual ways of thinking in order to start “thinking like a programmer.”

In Knowles’ view, adult learners have a task- or problem- centered orientation toward learning, as opposed to the subject orientation associated with the education of children. This has important implications for the current research in that learning how to program is often undertaken for instrumental reasons; that is, a student usually learns to program as a necessary step toward some other goal, not as an end in itself. Deciding whether to create LP/DE courses with an instrumental (as opposed to subject-based) orientation is an important aspect of course development.

Activity Theory

Activity theory, which has as its philosophical roots the works of Marx, seeks to "overcome and transcend the dualism between the individual subject and objective societal circumstances" (Engestom and Miettinen, 1999, p. 3). It focuses on the tool-mediated processes of society which, when the objective and the motive do not coincide, are called *actions*. Thus, a hunter driving game away from himself but toward his clansmen is performing an action, because his motions (pushing the hunted animals away from himself) differs from his motives (catching and killing the animals).

Activity theory highlights collective practices, positing that such practices "are not reducible to sums of individual action" (Engestom and Miettinen, 1999, p. 11). On its face, activity theory seems to share a worldview with the cognitive science concepts of situated cognition and distributed cognition (Engestom and Miettinen, 1999, p. 8); however, Tolman (1999) presents an analysis of how activity theory differs from what he calls Anglo-American contextualism, by which he means behaviorist and cognitive psychology, as well as those philosophies which are based on similar worldviews. His main position seems to be this: Anglo-American contextualism places the subject in the center of the external world which includes, among other things, social relationships which themselves are manifestations of the others' contexts, including history which is captured within each individual as the sum total of their own experiences. He rejects this viewpoint: “But for the individual, the ‘historical-sociocultural milieu,’ that is, ‘society,’ is not a set of variables that can be subtracted from the focus of concern by being held constant. The individual *is* society manifested in a single organism ... If context is a collection of variables that influence the already existing individual, then it cannot be identical to society. Also, therefore, activity theory and contextualism are not saying the same thing with different words. They are fundamentally different theories because they are based on fundamentally different philosophies: Activity theory is a consequence of classical German philosophy; contextualism is one of the many natural offspring of British empiricism” (Tolman, 1999, pgs 82-83).

To say that an “individual is society” is to posit that, were it possible to separate the individual from all remnants of society, there would be nothing left. Not all activity theorists seem to consciously posit this “empty core,” as can be seen by references to “a need for an approach that can dialectically link the individual and the social structure” (Engestrom, 1999, p. 19). If the concept of *the individual* and *the activity* are simply two archemidian points of view for analysis (see, for example, Engestrom, 1999, p. 32), then there is nothing dichotomous in these notions; however, those who reject the possibility of the individual are claiming more than this.

The current research draws from the spirit of activity theory in the lineage of Engestrom in that it highlights the student—instructor—other students relationship, but it does not discount the existence of the individual student. The individual student is built into the model primarily on a cognitive basis.

Cognitive theory

This study takes a primarily cognitive orientation with regard to theories of learning, as laid out in general in Matlin (1994) and in more detail in Pylyshyn (1984) and Newell (1990). Most of the learning-oriented articles cited in this paper employ a cognitive approach, and thus much of the research draws heavily from the information-processing model of cognition. The information processing model looks at the mind of the learner as a series of internal states which can be affected by the input of the senses and in return can affect the outside world (see, e.g., Matlin, 1994 for a detailed discussion of this approach).

This paper also assumes that a major goal of LP/DE is the student's ability to transfer knowledge and skills from one context to another, and that this can be influenced by factors under the influence of the instructor: "The basic elements involved in transfer are thus the learner, the instructional tasks (including learning materials and practice problems), the instructional context (the physical and social setting, including the instruction and support provided by the teacher, the behavior of other students, and the norms and expectations inherent in the setting), the transfer task, and the transfer context" (Marini and Genereux, 1995, p.2).

A focus on a cognitive model of the student is not overly restrictive; cognitive approaches can work well in concert with other views of learning. An example of how cognitive theory can work with activity theory was presented above. Activity theory focuses on actions for which the object and the motive are not one and the same (Engestrom and Miettinen, 1999, p. 4) and which are mediated by the use of tools. Activity theory wants to analyze "complex interactions and relationships … [a] theoretical account of the constituent elements of the system under investigation is needed ... [the] minimum elements of this system include the object, subject, mediating artifacts (signs and tools), rules, community and division of labor" (p. 9). In this respect, activity theory shares attributes with systems theory and also with the cognitive theoretical notion of situated cognition.

As an example of the crossover between cognitive systems theorists and constructivists, Luppicini (2002) discusses second-order cybernetic modeling, which "views knowing as a process of continual construction that maintains itself in the presence of (enabling or frustrating) perturbations from the medium in which it resides" (p. 93). Importantly, Luppicini adds that "knowledge is embedded in a circular social practice that involves thinking and acting beings"

(p. 93).

The task and psychology of programming

Computer programming involves building step-by-step instructions to "explain" to a computer how to implement a certain process. It is a goal-driven and often hierarchically organized process, and can involve a great deal of complexity: "A program consists of a list of statements which accomplishes some goal. In addition, a program can often be analyzed into a set of chunks or modules. Each chunk (or module) consists of two or more prestatements and accomplishes some subgoal within the context of the program" (Mayer, 1985, pp. 98-99). Programming involves a large range of knowledge and skill, ranging from an understanding of the programming language and environment through to generalized problem-solving strategies and knowledge of the domain in which the program will be used (McGill, Volet and Hobbs, 1997, p. 236).

The study of programmers

According to Pennington (1985), the study of computer programming and computer programmers has been undertaken since at least the early 1960s. By 1982, however, "relatively little empirical evidence (had) accumulated to shed light on what people actually do when programming as opposed to what we think they might do" (p. 29). As of 1982, programming had been compared to reading and writing prose, playing chess and "general problem solving." These various ways of viewing the task of programming are not necessarily mutually exclusive, since the prose analogy focuses on working from a program backwards to interpretation, the planning analogy looks at programming as a forward-chaining stepwise process, and the chess playing analog looks at programming as a form of expertise with knowledge “chunking” (p. 31). As of that time, there was no accepted notion of the basic unit of analysis with respect to the cognitive aspects of programming, it was still unclear even if this basic building block was function or data, and the notions of surface versus deep structure were not well differentiated.

During the 1980s there was active interest in studying the task of programming, probably driven by the intense economic value of computer programming during that time. This interest waned during the 1990s, although there has been a steady (though not large) contingent of researchers in this field. Two organizations which are fundamentally interested in this subject are the Special Interest Group on Computer Science Education (SIGCSE) of the Association of Computing Machinery (ACM) and the Psychology of Programming Interest Group (PPIG). Journals that periodically feature papers on this subject, in addition to the journals put out by the above organizations, are *International Journal of Human-Computer Studies*, the *Journal of Computing Sciences in College*, and occasionally in the journal *Cognitive Science*.

Most analyses of programming behavior since the mid-1980s have come from a cognitive perspective. Computer programming is very cognitively taxing:

“programming normally requires the co-ordination of multiple representations ... experienced programmers, when comprehending code, are able to develop a mental representation that comprises these different perspectives or information types, as well as rich mappings between them” (Romero, Lutz, Cox, R., and du Boulay, 2002, np). There are two aspects of code generation: “one in which the external structure (program code) is created from the internal (cognitive) structure which represents the problem requirements, and an inverse process in which the internal structure is recreated when necessary from the external structure” (Davies, 1991, p. 563).

Models and metaphors

Some of the most fruitful approaches to the study of programming have been based on seeing programming (a) as abstraction, (b) as linguistic process, (c) as planning, and (d) as attention investment. Each of these is described briefly below.

Programming as abstraction: Programming involves abstract thought; sometimes a great deal of abstract thought. All programs share the attribute that the user is not manipulating a particular thing, but instead specifying behavior to be performed at a future time and/or in particular circumstances (Blackwell, 2002, np). This can cause a strain on a programmer's short-term memory; working memory “overload” occurs when the subject has too much information or too many manipulations are required at any one time (Mancy & Reid, 2004, p. iii).

Based on this, some have proposed that a subject's working memory capacity may be an indication of his or her aptitude for programming.

Programming as linguistic processing: “Programming involves the manipulation of objects described in terms of a specific notation” (Davies, 1994, p. 722). All programming involves abstract symbol manipulations, in many ways like linguistic processes. Some programming languages look quite a bit like natural languages (e.g., COBOL), and others don't look like natural languages at all (e.g., APL). The programmer must be able to manipulate complex situations through the abstraction of indirect effects and the use of notational devices (Blackwell, 2002, np). This aspect of programming has led to the study of programming languages as an important area of research in the psychology of programming. In fact, some (see Anderson, 1983) propose that the structure of programming languages can be seen to mimic the structure of the human mind.

Programming as planning: The computer program is an imperfect correspondence between the programmer's internal plan and the outside world (Davies, 1994, p. 718). As a goal-directed process, an aspect of program assessment is the correspondence between the results of executing a program and the goal as defined in the specifications, the development of which are themselves problematic. Programmers, *qua* programmers, are not able to control the vagaries of specification errors (Ko and Myers, 2005, p. 43). However, a non-trivial aspect of both the task and the study of programming is an analysis of programming errors (or “bugs”), including how they occur and how programmers find and fix them (“debugging”).

Programming as attention investment: Blackwell (2002) proposes a model of programming using an *attention investment* approach: decisions to program something are made based on a calculation of the advantage gained by programming a repetitious task versus the mental cost involved in developing the program. This notion can even be applied to simple programming tasks such as programming a VCR.

<u>Novices and experts</u>

One of the most fruitful (for the purposes of this study) distinctions made in the literature on programmers is that between novices and experts. An important result from this research is that *novices and experts do not program the same way.* For example, an expert programmer usually will build a skeleton of a program, then go back in successive iterations and fill in the details; conversely, a novice will tend to write a program sequentially (see Davies, 1991, p. 553 and Lane and VanLehn, 2003, p. 150). Experts also read programs in the order they are likely to be executed (which is not sequential) whereas novices read sequentially (Davies, 1991, p. 561). More experienced programmers switch back and forth between the program and the output more often than novices (Romero, Lutz, Cox, and du Boulay, 2002, np). It is also theorized that expert programmers have a more highly structured schema representation of programming problems which allows them to more effectively search for an appropriate problem representation (Davies, 1994, p. 706): “One of the outstanding features of expertise in programming ... appears to be the particularly opportunistic nature of preferred cognitive strategy ... the programming activity can not be characterized as purely sequential; rather, it might be better construed as consisting of bouts of activity, each involving the creation of code fragments. These fragments are, in turn, continually reevaluated and modified in respect to the particular goal or subgoal currently under consideration. In addition, the development of code may be postponed at any time in order that the programmer might direct his or her attention to other goals or subgoals, possibly in response to the recognition of previously unforeseen interactions between code structures” (Davies, 1991, p. 565).

Expertise in programming is developed by a “knowledge compilation process” and/or “effectively utilizing previously externalized information” (Davies, 1994, p. 722); that is, an expert programmer is someone who has turned some number of complex tasks into automated steps which require a smaller amount of cognitive resources than they otherwise would. Many errors during programming occur because of a breakdown in the automated processes that the programmer employs (Ko & Myers, 2005, p. 51), and a novice's *fragile knowledge* of the details of programming is the cause of most student programming errors (Ko & Myers, 2005, p. 54).

According to Davies (1991), some have suggested that expertise in programming consists in the development of more and more models of program solutions, however the evidence for this has been equivocal and may be the result of “post-hoc rationalizations of the programmer's behavior” (p. 550). Instead, he argues, expert programmers act in a qualitatively different manner as compared to novices. For example a programmer's strategy changes from depth-first to breadth-first as the programmer develops expertise, and both experts and intermediates generate more plans than novices (Davies, 1991, p. 567).

It is an open question whether a student needs to go through a novice stage or if novices simply are often not taught appropriately. In other words, should introductory students be taught to mimic the programming behavior of experts, or must they first learn to program in the “novice mode” before they can develop expertise?

When is programming worth the cognitive cost?

Programming is a complex mental process which taxes cognitive processes, but these processes can be “packaged” into repeatable scripts which can then be used when appropriate with a lower cognitive cost. Still, the decision to program something, whether it is a VCR or a nuclear missile, comes with it a series of choices with respect to what will be gained and what it will cost in time and mental effort. Novices have to work harder than experts to produce a working program, and this might explain the high drop-out rate in programming courses. Whether students can “skip over” the novice stage and learn to program as experts from the start is an open question.

Teaching and learning programming

There is a wide variation in the ability of students to learn to program, and “there is a need to understand the fundamental causes that make [advanced] learners different from others, and the same applies for those students who fail to understand even the basics of the subject” (Mancy & Reid, 2004, p. ii). This section reviews the literature on the mechanisms of learning to program and the attributes of students with regard to their ability to acquire this skill.

How is programming learned?

The most common theory of how programming is learned is probably the “schema accretion” model, wherein the student builds up a “toolbox” of approaches to how to solve particular classes of problems. However, Davies (1994) argues for the importance of *knowledge restructuring* in the development of expertise. He offers empirical evidence for the changes in the structure of programming knowledge that develops with expertise. Another theory is that expert programmers have more complete schemata than novices (not just more, but fuller). Novices tend to focus on the key words in the problem statement rather than the deep structure of the problem (Davies, 1994, p. 706). Novices also tend to work in the order of the schema that they are using (Davies, 1994, p. 707). Mayer (1985) proposes that learning to program includes “acquiring a mental model of the underlying (computer) system ... a mental model is a metaphor consisting of components and operating rules which are analogous to the components and operating rules of the system." (p. 90)

In apparent contrast to those who theorize programming in a knowledge-centric manner, some see learning to program from the standpoint of *procedural planning*. For example, Mayer, Dyck and Vilberg (1986) found that training in appropriate procedural models may enhance students' ability to learn programming: "Students who were given pretraining in the output of English in procedures learned [programming] much faster than those with no pretraining ... a straightforward conclusion is the procedure comprehension is a component skill in learning [to program], and this can be taught to novices" (p. 609).

Student attributes

Several studies, mostly performed during the late 1980's and early 1990's, have identified various individual student attributes as significant with regard to learning to program. An empiricist view of learning to program says that the student must have appropriate previous experiences in order to learn to program; a nativist view says that the students differ innately in their ability to learn to program (Mayer, 1985, p. 121). Ability in programming has been linked to math ability, field independence and high capacity short-term memory.

However, the task of programming is not monolithic; different types of programming require different attributes. For example, learning modern object oriented programming requires different cognitive tasks than the older, procedural, programming approach (Romero, Lutz, Cox and du Boulay, 2002, np). At least since the 1970s there have been tests commercially available which purport to measure a person's aptitude for programming; however, these have not been widely used. Instead, a self-selection process has been the norm: if a student was able to persevere through a programming-oriented curriculum, the expectation has been that s/he was able to program reasonably well. Unfortnately, this self-selection process loses effectiveness when grade inflation pressures allow untalented students to obtain good grades in a programming class.

Process attributes

Several aspects of the process of learning to program have come to light through research. A simple synopsis of this research is as follows: Students learn to program through a consecutive series of assignments which help to engender in them a mental model, consisting of either a knowledge structure, a series of plans, or both, which are then used to solve more complex problems. A “good” mental model is one that helps novices learn to program better (Mayer, 1985 , p. 91).

The environment in which the learning occurs can either aid or hinder learning, based on familiarity and cognitive load requirements. Important environmental variables include the programming language used, the user interface, context, and the lab environment. For example, novices are often confused by the varying contexts in which the same programming language statement can be used (Mayer,1985, p. 98). Mayer also found some evidence that "the particular formalism used for expressing procedures may affect the relative difficulty of encoding particular types of transactions" (p. 120). Ko and Myers (2005) has a similar finding; a student must master the semantics of a programming language before s/he will be able to predict the outcome of a program and therefore effectively write programs (Fay and Mayer, 1988, p. 63). These results may point toward a programming language version of the linguistic relativity hypothesis (see Shaffer, 1996 and Shaffer, 1997).

The user interface used to develop the programs during a course can have a significant effect on learning. A balance needs to be struck between two competing priorities: (1) using an interface which enables the student to learn without undo cognitive strictures, versus (2) using an ecologically valid interface which will enable the student (especially the vocational student) to actually perform programming tasks in a non-educational environment. As already mentioned, certain user interfaces can be shown to be too mentally taxing for a novice programmer (Davies, 1994, p. 706): "Research has demonstrated that many students have difficulty acquiring the semantic structure of the programming environment" (Fay and Mayer, 1988, p. 63), although repeated exposure to the environment lessens this difficulty (i.e., the student can eventually "get used to" the user interface).

Finally, some (e.g., McBreen, 2001) propose that programming is best learned via a practical apprenticeship, which Jakovljevic (2003) describes as "based on three phases: observation, scaffolding and increasingly independent practice" (p. 309). Thus, programming is looked upon as a *task to be performed* more so than as a *field to be learned*. However, it would be wrong to think that programming can be taught in a strict behavioral fashion; there is definitely a cognitive content to computer programming which must be nurtured in order to develop real mastery. For example, Mayer (1985) demonstrates that having an appropriate mental model of the computer enables students to transfer what they have learned to new programming situations.

Is it possible to teach programming?

The notion of learning to program involves learning how to solve some example problems in order to be able to solve analogous problems at some time in the future. However, such transfer is rare even when the problems are formally identical (Mayer, Dyck &Vilberg, 1986, p. 605). Jakovljevic (2003) points out that "it is generally accepted that it is difficult to convey the method for transforming an abstract statement of a potential computer application into a working problem statement" (p. 309), and that "a teacher should establish a sequence of instructions that will lead learners to independent thinking" (p. 313); however, she does not offer any specific instructions on how this can be done.

Clearly programming can be learned; however, this does not imply that it can be taught. Perhaps those students who will "get it" would do so anyway, and those who do not will not. If programming can not be “taught”, then what is the role of the instructor in a programming class? Lehre, Guckenberg & Sancilio (1988) suggest a model of the programming instructor as a “moderator of novelty ... an instructor introduces discrepancy to instigate systemic reorganization” (p. 81). The instructor can also act as a filter for the student, in order to help differentiate the important information from the “noise” (Mancy & Reid, 2004, p. iii).

What should be taught?

There is a dichotomy with regard to the notion of what is to be taught in an introductory programming course, although few authors suggest one or the other extreme. The dichotomy is between discursive learning and programming practice. Although the goals of most introductory programming courses are nominally the same, there is a disagreement over what this entails – is programming best taught as a task or as a knowledge-based activity? As an analogy, it might be possible to teach students how to solve integration problems in calculus without ever explaining the semantics of the symbols. However, one could reasonably ask whether the student was learning calculus or not. It is possible to see a programming language as a formal system without a physical incarnation and to teach a student to program a certain way whenever a particular situation is encountered. However, some would argue that without a deeper understanding of what the programming language constructs symbolize, the student will at best only become a sort of programming language typist; that is, the student will not gain a deep knowledge of the craft.

Some have suggested explicitly teaching programming schemas (see Davies, 1994 for some references); for example, Lane and VanLehn (2003) found that students who were explicitly taught planning strategies “committed fewer structural mistakes ... and exhibited less erratic programming behavior” (p. 148). But Davies finds against this notion, since developing these schemas is a primary skill needed in expert programmers. Mayer (1985, p. 106) found that students explicitly taught a mental model of the computer were able to more creatively solve novel programming problems then those who were not explicitly taught the model; this was especially true of lower-ability students. Thus, if we want programmers who can creatively solve problems, it seems that we must teach students the deeper meaning of the symbols used in programming.

However, is this level of creativity needed in all programming? As mentioned above, some level of programming activity is inevitable for almost everyone in twenty-first century Western societies. Does the checkout clerk who must execute a sequence of keystrokes at the beginning of a shift need to know the inner workings of the inventory management system? Does a teacher trying to record a television program for her class need to understand the inner workings of a DVD recorder? These types of “programming” are common occurrences but are also the source of frustration for many people, perhaps simply because they do not understand the basic notions behind writing a program. Fay and Mayer

(1988, p. 73) found evidence that novices can better acquire programming semantics if they are first given experience with procedural design in English – perhaps the primary problem in learning to program is the ability to create step-by-step processes, and not in the particulars of a programming language. Lane and VanLehn (2003) propose that students be required to submit pseudocode (an English-like description of an algorithm) for assignments before being allowed to proceed to the programming step.

Prescriptivity

A fundamental issue in the study of teaching programming is the question of exactly what is to be taught; said a different way, the question is: Given two programs, what criteria are used to determine that one is better than another? Anderson and Skwarecki (1986) claim that "we need to incorporate precise and realistic models of how students should program and how they actually program" (p. 842), but whose version of this shall we accept? Programs are often seen by programmers as embodying a certain beauty and elegance which, as aesthetic qualities, are not able to be objectively scored. In this way, programs may be seen to be very like written compositions – some variation in style is expected, and not everyone likes everyone else's style.

Even if the question of what makes one program better than another can be answered, does the notion of a program being "80% acceptable" even make any sense, or is the very nature of the acceptability of a computer program itself

Boolean in that a program is either acceptable or it isn't? Traditional education in the United States is based to a great extent on the notion of partial credit, but what good is a program that “mostly” runs a nuclear power plant or space shuttle mission? This may call for the use of a mastery learning model (Gentile and Lalley, 2003). However, a focus on “answers” can get in the way of the actual goal of the task, that is developing internal models of the domain. (Lehre, Guckenberg, Sancilio, 1988, p. 96).

How should class time be spent?

There seems to be general consensus that the best way to teach programming would be the one-on-one tutorial (Clancy, Titterton, Ryan, Slotta, and Linn, 2003, p. 135); however, this is rarely economically feasible. In the United States, the two most common approaches are in-class lectures and labs; some courses are taught entirely via lectures (requiring the students to work the programs out on their own), and some are taught entirely in a lab environment, and many classes are taught in a hybrid environment. A small but growing approach is teaching programming via distance education.

Early notions of teaching programming focused on self-discovery, but research in the mid-1980s indicated that guidance is needed (Mayer, 1988, p. 4). Given that students must have an appropriate mental model in order to program well (Mayer, 1985), there is a need to impart discursive knowledge to the programming student. This could be done via lecture or readings, and thus the choice of how this information is conveyed is a pedagogically tactical one. According to Davies (1994), this choice is critical because the development of the structured representation of the expert programmer “may not simply arise via the control of a basic psychological mechanism such as knowledge compilation” (p. 703), but that “specific forms of training can give rise to (the) knowledge restructuring process” (p. 703), which he argues is needed to develop expertise in programming.

Clancy, Titterton, Ryan, Slotta, and Linn, M. (2003) suggest having students spend time on programming in a lab rather than listening to a lecture. When several students were found to be making the same mistake, the lab was interrupted for a brief lecture on the topic. Automated tools allow for continued monitoring of students' understanding and put the instructor into much more of a tutoring role. They found that “both the flexible pacing and the targeted tutoring served to force the students to keep up” (p. 133) and they found their worst students performed better than the average students in a traditional class. Shaffer (2005) describes a similar scenario.

Team and pair learning has received some attention, and most indications are that students learn to program better when they do it with others. Lehre, Guckenberg and Sancilio (1988, p. 96) found that students were more likely to decompose a problem into subproblems when working with a partner versus working alone. In addition, students need to have experience producing multiple solutions to a particular programming problem (Mayer, 1988, p. 11), which is more likely to happen in a group setting. Finally, reading other students' programs increases student program quality (Zeller, 2000, p. 91) – both the reviewer and the reviewed benefit. However, Kleinman and Entin (2002) found that students themselves did not think highly of working in pairs and groups (p. 217).

Lower versus higher ability students

The disparity in student abilities can cause class management issues, since “teaching to the center” may leave the higher ability students bored and the lower ability students lost. Additionally, Mayer (1985, p. 109) found evidence that treatments which helped lower ability students did not help those with a higher ability (in this case, ability was measured by SAT math scores). He also found that training lower ability students in appropriate mental models tended to enhance their performance in learning to program (p. 111).

One factor of individual differences between students is their working memory capacity (Matlin, 1994, p. 125), which may account for students' distribution of ability in mathematics. Designing methods which minimize memory load is one way that more students can be successful in learning to program: “Because working memory capacity predicted the acquisition of [programming ability], instructional methods should be analyzed with respect to the working memory demands they place on students” (Lehre, Guckenberg, Sancilio, 1988, p. 106).

The value of extended programming practice

Developing a deep knowledge of programming probably requires "prolonged periods of programming instruction" (Lehre, Guckenberg, Sancilio, 1988, p. 82); however, "the labor-intensiveness of grading programming assignments is the main reason why few such assignments are given to the students each semester, when ideally they should be given many more" (Cheang, Kurnia, Lim, Oon, 2003). In order to encourage programming time, Kleinman and Entin (2002) used a text that was a sequence of tutorials. "In addition to completing each hands-on tutorial and a series of small programs from the exercises, the students were asked to implement an individual project for each of the tutorials. They had to imagine a problem or situation that required a programmed solution, write a scenario, and then design the interface required to include the key ideas introduced in the new tutorial" (p. 213) Shaffer (2005) circumvented the problem of grading time by assigning programming tasks which were not graded but which figured prominently in the weekly exams.

What makes teaching programming successful?

The notion of teaching programming is problematic; however, it is clear that programming is learned every day by students in traditional classroom situations, and thus it must be possible. Jakovljevic (2003, p. 311) offers a list of four aspects that contribute to meaningful learning of programming in classrooms:

1. a collaborative learning environment with a mixture of analytical and holistic learning styles;
2. multiple dimensions in the programming context: the perceptual, affective and cognitive;
3. concrete representations of programming concepts and procedures through concept mapping; and
4. positive influence of cognitive apprenticeships and activity-based practice.

Although these features are based on the research, they may be too abstract to help the practitioner in implementing a specific introductory programming course with a wide range of student needs and abilities. Although the research cited is useful, finding a method of turning this research into practice is a significant challenge.

Technology aids for teaching programming

Unlike some disciplines, using computer technology is not optional in a computer programming class, and this adds an extra set of pedagogical decisions for the instructor. Zachary and Jensen (2003) suggest that distance educators have only exploited the communications aspect of computers, not the computational aspects (p. 396). This section focuses on the use of technological aids (primarily computer programs and systems) used in teaching programming, and discusses aspects of user interfaces for programming, often called interactive development environments or IDEs.

Various dimensions of the problem

When an instructor chooses a technological medium for use in teaching programming, s/he brings with it certain assumptions about the nature of teaching and learning programming. The following are important dimensions of technological decisions:

- Should a course use a development environment that is specifically designed for novice programmers or should it use tools that professional programmers use?
- If a “student” version is used, should it be intrusive or “hands off”? Should it help to guide the student actively?
- Should there be a tutorial aspect along with the programming interface, or should these remain separate?
- Should the interfaces be graphic or textual?
- Should the environment be multi-user or individual?

There is a tendency toward using the same tools in a class that are used by professional software developers, with the idea in mind that this is more ecologically valid. However, using professional tools to teach introductory courses is problematic; technology hurdles can be a major factor in drop-out rates (Kleinman & Entine, 2002), and the complexity of shifting through the myriad options in a professional IDE is likely to increase a student's frustration.

Various approaches to incorporating artificial intelligence techniques into programming tutors have been attempted, running the gamut from minimally intrusive systems (see e.g., Zachary and Jensen, 2003 or Shaffer, 2005) to systems which capture every student keystroke and interrupt the student as soon as an error is sensed (Anderson and Skwareki, 1986) or only require the student to fill in blanks in an almost complete program (Odekirk-Hash and Zachary, 2001).

Automated and peer review

One effective feature of learning to program is receiving outside review; using an IDE which allows students to collaborate as well as comment on each other's programs could be a major pedagogical advantage, although only a few have published results of this practice (e.g., Zeller, 2000). Using an automated grading system allows for more assignments to be given, giving the student more practice in programming (Cheang, Kurnia, Lim and Oon, 2003, and Shaffer, 2005). In both of these cases, technology can help to balance the competing goals of giving the students more programming assignments and minimizing instructor grading time.

The current state of learning to program via distance education

Although distance education has received much attention during the past ten years, there is a surprising dearth of research articles on teaching programming via DE. Most of the above cited sources are papers that are not specifically about DE programming courses; however, these and other papers on teaching programming can inform the current work. For example, as discussed in a previous section, there are a number of papers to be found on user interfaces for teaching programming – many of these are web-based interfaces which obviously could be used for DE courses, though they typically are not listed as such in the papers themselves.

However, special issues exist with regard to teaching programming via DE; Renaud, Barrow and le Roux (2001) point out that a DE organization has to make changes based on the knowledge that some DE students may take as long as 10 years to complete a degree. This means that changes to a curriculum have to be balanced with the needs of the existing students. In addition, changes to technology may occur which require changes to the students' home computers which further cause incompatibilities, etc.

Heo (2003) points out that “an increasing amount of programming-related courses is being offered in distributed education, a condition which directly facilitates an atmosphere of self-directed learning. However, the majority of lectures and code examples are functionally and visually static and remain organized around the delivery media rather than the knowledge representation and learning tasks of the student” (p. 152).

The most wide-reaching paper of those discussed here is Kleinman and Entin (2002), which “contrasts in-class and online teaching from both the student and instructor perspective based on two sections of Introduction to Computer Science, one taught in the traditional format and the other taught online” (p. 206).

They surveyed the students from both classes in the following areas: background, course expectations, course outcomes, course mechanics, and value of the course. They found significant differences in only a few areas: the DE students were significantly older, the DE students found the text to be more useful, the DE students felt the overall value of the course was higher, and the drop-out rate was higher than F2F courses.

Zachary and Jenson (2003) see much of the current DE course development as missing an important point: "The great strength of traditionally-taught classes is the real-time interaction that is possible between teachers and students. This strength is next to impossible to replicate in an online setting. Instead, online classes should exploit a different strength: the fact that every student who is involved in the class is sitting at a computer. It is essential to exploit the computational capabilities of the computer as an instructional medium" (p. 396).

Renaud and Le Roux (2001) list a series of problems encountered when teaching programming at a distance, and discusses how these problems were resolved within their department. The problems encountered included choice of programming language(s) to teach, the amount of student time spent on actual programming, paradigms to be taught, and required background for students entering the program. Through a series of departmental meetings the faculty were able to develop a solution by consensus, although it is unclear if these results can be universalized.

McGill, Volet and Hobbs (1997) discuss their analysis of individual student attributes which predict success in a DE programming course. They note that research indicates that first semester students are at high risk of dropping out and not returning to school, and thus they emphasize that extra support is needed for students who are at risk of withdrawing. “The crucial issue is to be able to identify early enough who are the students at risk. This can only be achieved through developing a better understanding of the factors associated with academic success, withdrawal or failure” (p. 240). They attempt to derive a predictive equation from their raw data, and find that the best predictor of course completion is a fairly complex combination of: (1) student confidence, (2) perceived relevance, (3) perceived importance of help from a tutor, and (4) entrance score. This equation correctly classified 82.5% of the students in the initial data sample. They then tested the equation on a “hold out” sample and found that it correctly predicted the drop-out status for 65.5% of the students in that sample, which was significantly better than chance. In the end they conclude that: “This study suggests that it may be possible to identify distance education students who are potentially at risk in their introductory computing course. The small amount of time and effort required to survey introductory students could be a valuable investment. Students with no background in computing, a mismatch between time needed and time available for practical work, lack of confidence, or low expectations of achievement could be quickly identified and contacted individually to discuss strategies for mitigating risk factors” (McGill, Volet and Hobbs, 1997, p. 253).

As this overview illustrates, special consideration should be given to programming courses within distance education curricula. To date, research in this area is lacking.

Closing comments on the literature review

As stated at the beginning of this chapter, this literature review provides background for understanding the model presented in the results chapter of this paper. Details of the variables incorporated into the model are left to a later chapter because they are in fact part of the data of this study. Some of the information provided in this chapter explains aspects of LP/DE which are outside of the model presented later – these are considered environmental variables which affect LP/DE but are not specifically modeled. Other parts of this chapter provide background information for the reader on variables which are incorporated into the model; these are presented for those readers who may not already be familiar with some aspects of LP/DE. The model developed in this paper stands on its own and does not rely on the information provided in this chapter.

Chapter 3

Methods

This study investigated the teaching and learning of computer programming via distance education (LP/DE) from a systems dynamics perspective using a much larger number of variables than is typically analyzed in educational research. As Altman (2001) has pointed out, "this coverage is critical because the validity of the model is jeopardized by every item of information processed by the problem solver but not represented in the model" (p. 190). The shear number of variables and the possible combinations of interactions requires a mechanism for communicating the model that is qualitatively different than the typical expository approach. Thus, system dynamics modeling is used as the primary means of organizing the data. The goal of the project was to create a deep model of the nature and scope of LP/DE, and this has required the use of a creative (but not unprecedented) approach to the study design.

There is a "chicken and egg" conundrum involved in developing a system dynamics model of an inherently complex process; the problem is that isolating variables yields ecologically invalid results, but systems models can not be created without these variables. A possible solution, and the one used in this study, is to create a model using existing published data and "best guess" thinking, the goal of which is to develop "version 1.0" of a model with the objective of testing and updating it as new information is uncovered. The advantage of this approach is that insights can be garnered from the model as far as it works and also as far as it does not. The goal of this project is to create the *Universalis Cosmographia* of LP/DE, which requires attaining broad coverage without sacrificing important details. The approach used is satisficing (discussed in chapter 2); that is, the solution sought is one that is *good enough* for the defined purpose, though perhaps not ideal. Even if the results are less than perfect, there is value in at least getting started.

Research methodology

The approach used in this study is amenable to that suggested by Toulman (2004), who emphasizes the need for description before theory: “Until the basic empirical or experimental facts are established in the human sciences ..., we are not in a position to develop theoretical explanations, and the pursuit of theoretical generalizations is premature” (p. 61). He suggests that we step away from over-intellectualism and instead that we focus on *practice*. The two methods used in this study are meta-ethnographic analysis and qualitative study; a meta-ethnographic analysis (Noblitt & Hare, 1988) of existing research in relevant subject areas was undertaken in conjunction with a qualitative investigation of the students and instructors in four LP/DE programs in Pennsylvania community colleges.

The purpose of this study is to create a pragmatic, descriptive model – no claim to universality will be made; the usefulness of this model is as an organizational device for researchers and those interested in implementing a LP/DE program – the expectation is that the individual details of the situation will vary from case to case.

Qualitative study of students and instructors

A qualitative empirical study of learning to program via distance education in Pennsylvania community colleges was performed during the Fall of 2005. The selection of Pennsylvania community colleges was a convenient sample, which is justifiable based on the fact that no generalizability will be proposed. However, the sample is thought to be rich for the following reasons: (1) based on interviews with teachers, it seemed that the range of backgrounds and experience of students in these courses is quite diverse; (2) as the product of a Pennsylvania community college, I believe I have some understanding of the *zeitgeist* of these institutions, and (3) community colleges are in an advantageous position to offer LP/DE, especially serving the work-related and general education student (both of which are considered key targets for these courses into the future).

The fourteen members of the Pennsylvania Commission for Community Colleges were investigated to see if they offer LP/DE courses; five institutions were identified. These institutions were contacted via emails sent either to department chairs, course instructors, or both. Of these five, four institutions agreed to participate. Six instructors from the four participating institutions were identified and agreed to participate. One was subsequently dropped from the study due to the instructor/participant’s lack of interest. Three of these instructors were teaching strictly via DE during the semester during which data was collected; two were teaching both DE and face-to-face (F2F). Qualitative data on these participants were collected via face-to-face interviews and probing questions delivered via email approximately every two weeks during the semester.

A web-based survey was administered to the students at the beginning of the target courses to collect basic information about the students' background and interests. This survey was used to identify eight students for inclusion in the qualitative study, with the goal of selecting subjects with a wide range of backgrounds and life situations. Once the students signed on to the study via an email reply to the informed consent agreement, they were asked detailed probing questions about their experiences throughout the semester. Student subjects were paid $50 to participate.

The detailed answers supplied by both the instructors and the students were used as data in the development of the model. Execution of the study was uneventful, though it required a good deal of organizational skill to contact and re-contact the subjects in order to obtain responses to the probing questions.

Meta-ethnographic analysis of other studies

The approach taken in this part of the study is somewhat non-standard and warrants some explanation. The method is drawn primarily from Noblit and Hare

(1988), but differs enough that some justification of the approach is in order.

Ethnography

In its most basic form, ethnography is "the art and science of describing a group or culture" (Fetterman, 1998, p. 1) and, although it is characteristically carried out in a journalistic fashion, “there is a sense in which all social researchers are participant observers; and, as a result, the boundaries around ethnography are necessarily unclear. In particular, we would not want to make any hard-and-fast distinction between ethnography and other sorts of qualitative inquiry” (Hammersley and Atkinson, 2003, p. 2). Most notably for the present study, “ethnographic research employs three kinds of data collection: interviews, observation, and documents. This in turn produces three kinds of data: quotations, descriptions, and excerpts of documents, resulting in one product: narrative description. This narrative often includes charts, diagrams and additional artifacts that help to tell ‘the story’… Ethnographic methods can give shape to new constructs or paradigms, and new variables, for further empirical testing in the field or through traditional, quantitative social science methods” (Genzuk, 1999).

Ethnographies attempt to create a “thick description” of a social phenomena, and “although one starts any effort at thick description … from a state of general bewilderment as to what the devil is going on … one does not start (or ought not) intellectually empty-handed. Theoretical ideas are not created wholly anew in each study; as I have said, they are adopted from other, related studies" (Geertz, 2000, p. 27).

The current study is itself an ethnography in the manner described above; however, the subjects of this study are not people, but interpretive studies by and about people, and thus a better designation is that of meta-ethnography.

Meta-ethnography

According to Noblit and Hare (1988), a "meta-ethnographic approach enables a rigorous procedure for deriving substantive interpretations about any set of ethnographic or interpretive studies" and "a meta-ethnography can be considered a complete study in itself" (p. 9). Pielstock (1998) explains meta-ethnography as "a method to conduct an interpretive synthesis of qualitative research and other secondary sources as a counterpart to meta-analysis for quantitative research" (np).

A meta-ethnography is different from a meta-synthesis in that it is interpretive rather than aggregative, and "must be driven by the desire to construct adequate interpretive explanations" (Noblitt and Hare, p. 11). Noblitt and Hare's notion of "preserving the sense of the account through the selection of key metaphors and organizers" (p. 13) fits well with the idea of creating system dynamic models of the subject under study, and, as argued earlier, a significant use of educational research is to develop models and metaphors for the practitioner.

Meta-ethnography is used "to synthesize information regarding a phenomenon … that has been extensively described in a variety of other sources including quantitative studies, qualitative studies, and professional expertise" (Pielstock, 1998, np). However, meta-ethnographies differ from standard literature reviews: "The study-by-study presentation of questions, methods, limitations, findings, and conclusions lacks some way to make sense of what the *collection* of studies is saying. As a result, literature reviews in practice are more ritual than substantive accomplishments" (Noblit and Hare, 1988, p. 15).

A difficult problem in educational research is being able to apply the results of research studies in the day-to-day machinations of the classroom. The advantage of syntheses is that they guide but do not prescribe: "They enable us not to predict but to 'anticipate' what might be involved in analogous situations; they help us understand how things might connect and interact" (Noblit and Hare, 1988, p. 18). This is a different approach to abstraction than the more common meta-analysis, in that a meta-ethnography is an abstraction of summaries rather than a summary of abstractions; a meta-analysis deceivingly seems more "concrete" because the abstractions are done earlier in the process, i.e., during the design of the quantitative studies. In Noblit and Hare's approach, the studies that are summarized are themselves in many cases ethnographies or case studies, which means that there is little abstraction done in the base study design.

The present study differs from Noblit and Hare's approach in that it includes both quantitative and qualitative studies in the data; however, "as an interpretive methodology, meta-ethnography is not limited to synthesizing strictly comparable studies as meta-analysis is" (Pielstock, 1998). The goal of this project is the same as other meta-ethnographies; that is, to develop a coherent and useful model for applied use. In addition, the number of studies synthesized in the present paper is larger by an order of magnitude than that of other meta-ethnographic studies. The result is meant to be seen as metaphoric: "by treating these abstractions as metaphoric, we prevent premature closure on their meaning" (Noblitt and Hare, 1988, p. 33).

It might seem that this opens the door to those very "vague and superficial analogies" that von Bertalanffy was concerned to defend against (von Bertalanffy, 1968, p. 35); however, Noblitt and Hare (1988) defend against this view by delineating an historical argument for "three basic criteria for the adequacy of metaphors in social science: economy, cogency, and range" (p. 34). They also propose the following quality criteria for interpretative synthesis: clarification, resolution, consistency, parsimony, elegance, fruitfulness, and usefulness (p. 16). The current study uses a methodology very similar to Pielstock (1998), who lists the following "criteria of soundness" for a meta-ethnography: credibility, transferability, dependability and triangulation.

Noblit and Hare propose the following stages of a meta-ethnographic study: (1) getting started; (2) deciding what is relevant; (3) reading the studies; (4) determining how the studies are related; (5) translating the studies into one another; (6) synthesizing the translations; and (7) expressing the synthesis. The meta-ethnographic portion of this study was performed using this approach, except that step five could not be fully performed due to the shear volume of the potential combinations (3160). Specifically, this study is a species of what Noblit and Hare call a “lines of argument synthesis” (p. 62), the goal of which is “to discover a 'whole' among a set of parts” (p. 63). This is similar to the notion of argument-based validity as described by Kane (1992). Noblit and Hare go on to note that “the 'integrating scheme' that results must encompass all the data and also be 'open-ended' enough to allow consideration of new data and conceptual levels” (p. 63). This goal is captured by the system dynamics modeling approach taken in this paper.

Lastly, this research is appropriately termed a meta-ethnography because the focus is on the descriptions, results and conclusions of the papers studies, and not primarily on the quantitative data itself.

Getting started

This study was, in essence, started almost fifteen years ago. Since then I have been a student of computer programming, cognitive modeling, artificial intelligence, educational psychology, adult education and distance education. The selection of this topic is an natural outgrowth of my background in these areas. In consultation with my advisor on this project, I developed the basic approach to the study and “got started.”

<u>Deciding what is relevant</u>

The next step in implementing this project was selecting appropriate published studies for the analysis. This is separate and distinct from the literature review as presented earlier, which is presented as background only. This stage of the study required that a manageable number of published sources be selected for analysis; although the study is wide-ranging, it was necessary to limit that range in order to avoid the risk of never completing the work. Thus, although this study is informed by theories of educational administration and economics, the study was focused on published sources from the perspective of the cognitive, pedagogical and practical considerations of LP/DE. A review of published sources uncovered the following sources of information:

- The works of Mayer (see references) and Davies (see references), both of whom published extensively on the cognitive aspects of programming.
- Journals from the Institute of Electrical and Electronics Engineers (IEEE) and Association of Computing Machinery (ACM); especially the ACM Special Interest Group on Computer Science Education.
- The *Psychology of Programming Interest Group* (www.ppig.org).
- The specialty journals *The Journal of Computing Sciences in Colleges* and *Computers in Education*.
- Additional sources uncovered via literature searches on the following databases: Compendex, Dissertation Abstracts, Ebsco Host, Wilson Educational Abstracts, ERIC, Elsevier ScienceDirect, Kluwer Online,

ProQuest, PsycInfo, and Google Scholar.

The first pass through this selection process produced a list of 130 papers on appropriate topics. This list was culled to 80 resources; resources were eliminated if they: (1) contained heavy emphasis on technology over pedagogy, (2) were "work-in-progress" reports without any detail and/or conclusions, (3) were primarily about computer interfaces, or (4) were primarily "musings" on the subject. This completed step two of Noblit and Hare's seven step process.

Reading the studies

Two passes were made through the readings: The first pass was used to identify themes and metaphors which seemed central to LP/DE. The second reading of the material was used to create the system dynamics model. These readings were performed coincident with the next three stages.

Determining how the studies are related

While reading the studies during the first pass, a three dimensional grid was created (see figure 3.1) as an organization device. Themes were identified and entered into a spreadsheet, indicating where they had been found and categorizing them into one or more of the broad categories displayed across the x-axis. These categories changed and matured during the initial readings. At the end of the first pass, the resultant spreadsheet was sparsely populated, but this structure allowed for ease of access of the data for the next stage of the study.

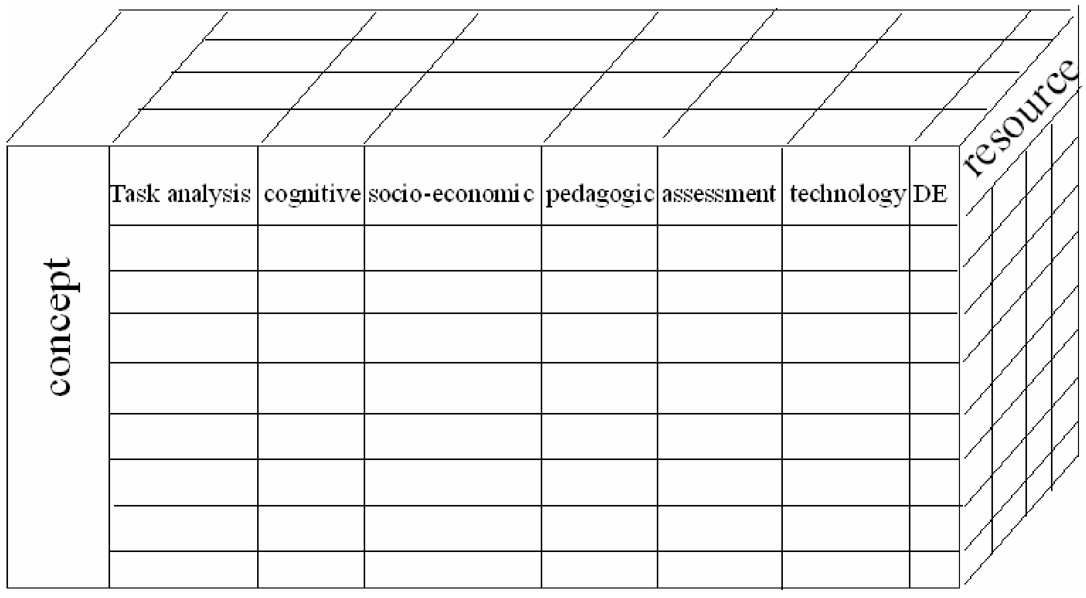

Figure 3-1: three dimensional spreadsheet of resource attributes

<u>Translating the studies into one another</u>

By analyzing the (rather large) number of concepts that existed in the spreadsheet at the end of the first reading, various concepts were merged into one another and patterns were discerned. However, due to the large number and wide diversity of papers examined, this step from Noblitt and Hare (1988) would have required 3160 combinations and thus could not be exhaustively performed. Instead, data from the various sources were combined in iterative stages.

<u>Synthesizing the translations</u>

During this stage, a second pass through the studies was performed, resulting in the system dynamics model presented in the next chapter.

Development of this model proceeded as follows:

1. Select the (next most) central concept based on an analysis of the data. This was informed by simple counts of references, but is essentially an interpretive step, requiring the use of contextual judgment.
2. Create a summary of the variable from the sources (both published sources and the qualitative study results).
3. Incorporate the selected concept into the model. This involved creating a new variable in the diagram and then drawing connections between the new variable and any others already in the model.
4. If there are more variables in the list, go back to step 1.

Expressing the synthesis

The unfortunate result of building models with many variables is that the resultant diagram can be overwhelming and/or difficult to understand:

> “If a complex diagram has results [more than 25 symbols, your audience] may be ‘turned off,’ saying that it is too complex for the time that have to study it or for a human’s ability to absorb its meaning. But if you show someone an aggregated diagram they may dismiss it as simplistic. If you show them both, they can see what has gone into the model and how simplification may be necessary to hold human credibility and interest. Then they may join you in modifications, calibration, testing

and revising" (Odum and Odum, 2000, p. 17).

Thus, expressing the synthesis in this case required breaking the larger model into smaller sub-models for easier understanding. This resulted in a third pass through the material in order to separate it into three broad categories (cognitive, pedagogical and practical). This enabled the model to be presented in a way which should be less overwhelming to the reader.

However, breaking the model into these categories is a departure from the basic notion of a system dynamics model, since it inherently constrains modeling the interconnections between concepts from different categories. Thus, the final step of this part of the study was to bring the variables back together again, using color coding to differentiate each of the three categories. In addition, explanatory "pop-ups" were added to each of the concepts in the diagram with the goal of making the model more useful for browsing.

Scope and limitations

As Jackson (2003) has pointed out, "it is the essence of an informal domain that the repertoire of potentially relevant considerations cannot be exhausted, so we can never be sure that our reasoning will not be invalidated by some consideration we have neglected" (p. 15), and thus it is with this study. Clearly there are aspects of LP/DE that have not been included in the model; some of the more obvious are economic and institutional organizational variables, but even within the purview of the domains covered there are bound to be omissions.

However, the study goals as stated are to create a suggestive and useful model, not a prescriptive nor a predictive one; for this reason these limitations are not viewed as diminishing the value of the project.

Chapter 4

Results

This chapter presents the results of both the meta-ethnographic study and the study of instructors and students in LP/DE courses in Pennsylvania community colleges. The chapter is broken down into three sections: the first section shows the results of the initial student survey; the second section presents a summary of the results of the studies; the last section shows the development of the preliminary model which was the principal outcome for this study.

Summary of the survey results

The first data collected in this study were the answers to the survey questions sent to students. There were a total of 36 respondents, and the results of most survey questions are presented below. It is important to note that the answers to the survey questions were used to select participants for the qualitative study, and no direct quantitative conclusions are being drawn. However, the data is reported here because it might be instructive in some cases and as such some discussion is included in each section below.

Age

Table 4-1 shows the range of responses for the ages of the students in the LP/DE classes studied. There is an obvious clustering around 18-21 and more than 30, indicating that people in their mid-twenties were not as likely to be taking an LP/DE course as those under 22 or over 30. The explanation might be that students 21 and under are undergoing their first post-secondary education in preparation for entering the workforce, whereas students 30 and above might be in the process of retraining e.g., for a career change or after a job loss.

```
+--------------+----------+
| age          | count(*) |
+--------------+----------+
| 18-21        |       12 |
| 22-25        |        6 |
| 26-30        |        1 |
| more_than_30 |       14 |
| under_18     |        3 |
+--------------+----------+
```

Table 4-1: Age distributions

```
+---------+----------+
| gender  | count(*) |
+---------+----------+
| female  |       16 |
| male    |       19 |
| skipped |        1 |
+---------+----------+
```

Table 4-2: Gender distributions

Gender

Table 4-2 shows the gender distributions of the participants. Historically, computer programming has been a male-dominated field. The relatively close gender distribution might indicate that there is a balance of genders in the study group, or it might indicate that females were more likely to volunteer for the study.

<u>Planned study hours</u>

Students were asked how many hours per week they planned to spend on the course. The data in table 4-3 is interesting in that 70% of the students anticipated spending less time per week on the course than the generally prescribed nine hours for a three-credit course. This result may indicate one or both of the following: (a) students thought that the DE course would be easier than a F2F course, and/or (b) students do not accept the prescription of nine (three in class plus six outside of class) hours per week for a three-credit course.

<u>Hours working</u>

Students were asked how many hours they would be working at a job while taking the course. Table 4-4 shows that over one-third of the students were working full time, and that most were working a significant amount of hours. One of the advantages of DE courses is the flexibility of scheduling for working students, and this result indicates that this might have been a significant factor in the students' decision to choose a DE course. This is borne out by the answers to the qualitative questions given by students later.

```
+---------------+----------+
| planned_hours | count(*) |
+---------------+----------+
| 0_3           |        6 |
| 3-5           |       10 |
| 5-7           |        9 |
| 7-9           |        7 |
| 9-12          |        2 |
| more_than_12  |        2 |
+---------------+----------+
```

Table 4-3: Planned hours

```
+---------------+----------+
| hours_working | count(*) |
+---------------+----------+
| 11_20         |        4 |
| 21_30         |        9 |
| 31_40         |        3 |
| 40_or_more    |       13 |
| none          |        7 |
+---------------+----------+
```

Table 4-4 Working hours

```
+------------------+----------+
| degree_program   | count(*) |
+------------------+----------+
| business         |        2 |
| computer_related |       21 |
| liberal_arts     |        2 |
| math_science     |        3 |
| no               |        8 |
+------------------+----------+
```

Table 4-5: Degree program

```
+------------------------+----------+
| reason_chose_this_class | count(*) |
+------------------------+----------+
| convenience            |        8 |
| cost                   |        1 |
| quality                |        6 |
| required               |       20 |
| skipped                |        1 |
+------------------------+----------+
```

Table 4-6: Reason for taking the class

<u>Degree program</u>

As shown in table 4-5, a little over half of the students were taking the course as part of a computer-related degree program. This is interesting because some have suggested that one reason for the bimodal distributions of grades in introductory programming courses is that students who are computer-oriented are mixed in with those who are not. In these cases, instructors may experience difficulties when “teaching toward the middle” because the middle is essentially empty.

<u>Reason for taking the class</u>

As indicated by table 4-6, a little over half of the students enrolled in the class because it was required. It is reasonable to assume that the rest of the students were continuing education students who thought that taking the course would better their lives in some specific way and, of those, most of them chose the DE course for convenience. However, a significant number of those who were not required to take the course did so because they thought the quality would be good.

<u>How good a programmer will you be?</u>

Students were asked to rank their impression of how good they might become at programming. This question was asked because some studies have shown that a student’s expectation of their ability is a good indicator of his or her final grade. It is not clear whether this is because of students’ strong self-knowledge or if it is a self-fulfilling prophecy. Table 4-7 shows that most students thought they would eventually be a “good” or an “excellent” programmer. This result might indicate self-selection into (a) programming in general and/or (b) a DE programming course.

```
+-------------+----------+
| how_good?   | count(*) |
+-------------+----------+
| excellent   |       11 |
| good        |       19 |
| passable    |        6 |
+-------------+----------+
```

Table 4-7 How good a programmer will you be eventually?

```
+----------------------+----------+
| confident_of_success | count(*) |
+----------------------+----------+
| certain              |       20 |
| expect_to            |        4 |
| hope_will            |        3 |
| pretty_sure          |        9 |
+----------------------+----------+
```

Table 4-8: Student’s confidence in their success in the course.

<u>Confidence of success</u>

Students were asked how confident they were of their success in the course; as shown in table 4-8, two-thirds expected to succeed or were certain of their success. However, one-third of the respondents were uncertain about their ability to do well in the course, indicating an interesting direction for future study: Why would one third of the students question their ability to succeed in this course?

```
+--------------------+----------+
| programming_before | count(*) |
+--------------------+----------+
| no                 |       18 |
| yes                |       18 |
+--------------------+----------+
```

Table 4-9: Previous programming experience

```
+------------------+----------+
| computer_comfort | count(*) |
+------------------+----------+
| fairly           |        7 |
| nervous          |        1 |
| uncomfortable    |        1 |
| very             |       27 |
+------------------+----------+
```

Table 4-10: Student comfort level with computers

Previous programming experience

Students were asked if they had previous programming experience; this might be in high school or elsewhere. As indicated by table 4-9, students were equally distributed on this question. This result is significant because several researchers have found that students with previous programming experience do significantly better in programming classes. Although this outcome might seem obvious, it has important social consequences in that students with more privileged backgrounds will then tend toward having a higher grade point averages upon graduation.

Comfort level with computers

75% of students reported a high degree of comfort with computers, as shown in table 4-10. What is of more interest pedagogically, perhaps, is the two students who indicated that they are uncomfortable or nervous with computers. Given the nature of the course (computer programming) and the nature of the delivery mechanism (web-based), it is a concern that these students were enrolled in this course at all.

```
+--------------------+----------+
| instructor_contact | count(*) |
+--------------------+----------+
| not_at_all         |        1 |
| pretty_important   |       10 |
| somewhat_importa   |        7 |
| very_important     |       18 |
+--------------------+----------+
```

Table 4-11: Importance of instructor contact

```
+-------------------+----------+
| attitude_to_class | count(*) |
+-------------------+----------+
| dreading          |        1 |
| like_any_other    |        4 |
| looking_forward   |       22 |
| nervous           |        9 |
+-------------------+----------+
```

Table 4-12: Attitude toward class

Importance of instructor contact

Over 75% of respondents believe that instructor contact was “pretty” or “very” important, indicating that these students expect tutelage and are not anticipating, for example, a computer-based training experience.

Attitude toward the class

Approximately one quarter of the class indicated that they were nervous about or dreading the course, as shown in table 4-12. Some researchers suggest that these students might be well served with some “ice breaker” activities to help alleviate their concerns.

```
+----------------+----------+
| math_important | count(*) |
+----------------+----------+
| neutral        |        4 |
| no             |        1 |
| yes            |       31 |
+----------------+----------+
```

Table 4-13: Is math important?

```
+--------------+----------+
| good_at_math | count(*) |
+--------------+----------+
| no           |        6 |
| sort_of      |       11 |
| yes          |       19 |
+--------------+----------+
```

Table 4-14: Math ability

Attitude toward mathematics

Tables 4-13 and 4-14 show that students overwhelmingly thought that mathematics is important, and 80% of them believed themselves to be “good” or “sort of good” at it. One of the most commonly reproduced results in the study of LP is that students who do well in math tend to do well in programming. In addition, 50% of the students reported having taken trigonometry, while 30% had taken calculus as well. Given that the students are enrolled in a community college, it seems reasonable to assume that most of the students who had taken calculus were continuing education students as opposed to being enrolled in an associate-degree program.

```
+-------------+----------+
| cryptograms | count(*) |
+-------------+----------+
| hate_it     |        1 |
| like_it     |        5 |
| love_it     |        5 |
| neutral     |       15 |
| never_did   |        9 |
| skipped     |        1 |
+-------------+----------+
```

Table 4-15: Attitudes toward cryptograms

```
+-------------------+----------+
| crossword_puzzles | count(*) |
+-------------------+----------+
| dislike_it        |        2 |
| hate_it           |        2 |
| like_it           |        9 |
| love_it           |       10 |
| neutral           |       11 |
| never_did         |        2 |
+-------------------+----------+
```

Table 4-16: Attitudes toward crossword puzzles

Like/dislike of various activities

Students were asked to rank their relationships to the cognitive activities of doing cryptograms, crossword puzzles, poetry and playing computer games. The results are shown in tables 4-15 through 4-18. Perhaps the most surprising result here is that approximately the same number of students reported “liking” or “loving” poetry as compared to video games; this certainly challenges stereotypes in this area. Future research into the relationship between an affinity for poetry and programming ability could be very enlightening.

<u>Computer resources</u>

As shown in tables 4-19 and 4-20, a large majority of respondents had access to their own computers with some sort of high speed Internet connection. Since success in an LP/DE program can depend very much on access to computer resources and, even though the respondents come from a broad range of socio-economic circumstances, it appears that such access is not an issue.

```
+------------+----------+
| poetry     | count(*) |
+------------+----------+
| dislike_it |        2 |
| hate_it    |        1 |
| like_it    |       12 |
| love_it    |       10 |
| neutral    |        7 |
| never_did  |        4 |
+------------+----------+
```

Table 4-17: Attitudes toward poetry

```
+-------------+----------+
| video_games | count(*) |
+-------------+----------+
| dislike_it  |        1 |
| like_it     |       12 |
| love_it     |        9 |
| neutral     |       11 |
| never_did   |        3 |
+-------------+----------+
```

Table 4-18: Attitudes toward video games

```
+-----------------+----------+
| computer_access | count(*) |
+-----------------+----------+
| family_share    |        2 |
| own             |       34 |
+-----------------+----------+
```

Table 4-19: Computer access

```
+---------------+----------+
| connect_speed | count(*) |
+---------------+----------+
| cable         |       21 |
| dialup        |        3 |
| dsl           |       12 |
+---------------+----------+
```

Table 4-20: Connection speeds

These survey results are enlightening and can be seen to indicate some areas of future research; however, the primary use of these survey results for the present study was to select a broad range of students for the qualitative study.

Summary of the results of the qualitative study

The second set of data collected for this study was qualitative, opened-ended interviews of eight LP/DE students selected from those who had responded to the survey. After obtaining informed consent, the subjects were asked to provide a brief biography; these are presented in appendix B. From these biographies it seems that the goal of diversity of subject backgrounds was met, which helps to explain the wide array and often contradictory answers obtained from the student subjects.

Results of qualitative study of students

Response levels were good if not ideal. After repeated entreaties most of the subjects responded most of the time. The results met expectations and were a good source of qualitative data for the main study; specific results related to each variable are presented below as they are integrated into the model-building process. As expected, no generalized conclusions could be or were drawn; in fact (also as expected), subject responses often contradicted one another.

Results of the qualitative study of instructors

Four instructors were surveyed approximately every two weeks throughout the semester. Although the existence and quality of responses were not consistent, enough instructors answered enough of the questions that useful information was gleaned. Again, the details of these responses are incorporated into the model building process documented later in this chapter.

Results of the meta-ethnographic analysis

This section describes in detail, for one variable (*cognitive load*), the process of deriving the elements of the preliminary model that is presented in later in this chapter. This section is meant to illustrate the process involved; reporting on each variable to the level described here would result in an unwieldy document and thus is not included. It is expected that this sample illustrates the process well enough to make the project transparent to the reader.

As described in chapter 3, the initial steps in the meta-ethnographic analysis were getting started, deciding what is relevant, and reading the studies; these processes are described in that chapter. During the initial reading, a grid was created into which new concepts were added and categorized as they were encountered. For example, the notion of *cognitive load* is explicitly discussed in van Merrienboer, Schuurman, de Croock, and Paas (2001) and Robbins (1999), and thus a “line item” was created for that concept. Also, additional support for the concept was found to be implicit in the concepts of working memory (Ko and Myers 2005, Lehre, Guckenberg and Sancilio, 1988) course design (van Merrienboer, Schuurman, de Croock, and Paas, 2001), and cognitive style

(McKay, 2000). Thus, *cognitive load* was seen to be a primary concept in the study of LP/DE and also seen to be linked to these other variables during the next phase, i.e., translating the studies into one another. It was also categorized as important to the cognitive, pedagogic and assessment factors of the study.

The final phase – expressing the synthesis – required attempting to capture and communicate visually how *cognitive load* "fits into the big picture." Doing this requires giving up a certain amount of precision, since it is probably true that every variable has an effect on every other variable in the model; however, a visual model with lines connecting every variable to every other one would not have communicative value. Thus *cognitive load* is shown directly connected only to three variables: cognitive style, stress and anxiety, and ability to perform the programming task. An argument can always be made to add or change any such link; this is part of the life-cycle of such a model – as critique and/or new information is absorbed, changes to the model must be made.

This explains the process of how the variables were identified and linked, together forming the preliminary model presented later in this chapter. The essence of these results is best understood by a close analysis of the model developed in the next section; it is here where all of the factors studied are brought together into a cohesive whole.

A Preliminary Model

As Rountree, Rountree, Robins and Hannah (2004) have said, “when the subjects under investigation are people, factors interact”; these researchers seek to “capture interacting factors in a natural and effective manner” (p. 103). Although the method used in the current study is different from that used by Rountree et al., the impetus is the same.

Learning to program via distance education (LP/DE) is a complex process that can not be captured via simple two-level diagrams. As discussed in previous chapters, system dynamics attempts to model the myriad interrelationships inherent in a complex system; unfortunately, doing so requires that the representation medium be sufficiently powerful to capture those relationships. This leads to complex-looking diagrams and can be daunting; such is the case with the resultant model of this study. In this section I attempt to break the model down into its component features without sacrificing the power of the model itself. A note of caution is in order: The explanation should not be confused with the model! The full model is presented toward the end of this chapter; the interim information is meant as scaffolding for the reader.

How the model was developed

Methods for developing system dynamics models are presented in Sterman (2000) and Odum and Odum (2000). Creating system dynamics models is essentially a creative process of iteration of paying attention to each variable and how it relates to the other variables in the system. The process is creative in that the model-builder must determine what is germane, what is not, which variables should be combined, and which should be separated. There is no (nor could there be an) algorithm for creating such a model, since it depends completely upon the intentions of the designer. (This assumes that computers do not themselves have intentions.) Thus, the purpose of the model drives its construction. In this study, the purpose of the model is explanatory and organizational; the model is meant to help explain and organize the multitude of variables involved in LP/DE. Such a model has the following advantages over mere textual description:

1. Visually the model indicates the breadth and depth of the subject under study.
2. Variables and relationships are specifically identified.
3. The dynamics of the domain, especially the loop-back features, are easier to recognize.
4. A computer representation of the model can be captured algorithmically and used to create simulations.

During its development, the model took many forms. At one point in the development, it became clear that some aspects of LP/DE had to be removed from the model and repositioned as environmental variables (outside of the control of the model under construction). This was necessary because of the size of the project undertaken; if this decision had not been made either the project would not have completed or the resultant model would have suffered from the same oversimplification that it was specifically created to counter. Thus, aspects of organizational dynamics and economics were eliminated, not because they are thought to be unimportant, but because the focus of the model became the cognitive, pedagogical and practical aspects of student/ teacher interactions in LP/DE.

How to read the model

The full model is meant to be studied as one would study a map of a city one intends to visit; it is possible that the reader has already visited parts of the city and/or heard about certain aspects of the city; or perhaps the city is completely unknown to the reader. In either case, reading the map ought to engender in the reader both an overall view of the structure of the city, as well as enable the reader to focus on particular parts of interest. Whenever the reader focuses on a certain aspect of the map (a neighborhood or perhaps a rail station), s/he is then able to get more detailed information by turning to the detail page on that feature of the city. The model presented in this chapter is meant to serve the same purpose for the visitor (instructor or researcher) to the field of LP/DE.

Model elements are ramified into cognitive, pedagogic, and pragmatic variables. The arrows indicate a relationship; for example an x → y relationship indicates that values for x cause changes in the values for y. It is important to note that these are not time-stamped elements as in, for example, a process flow chart. Referring to diagram 4-1, one should not read the diagram as “first there is a hierarchical schema structure, then there is an appropriate mental model / schema, then there is the ability to retrieve knowledge.” Instead the relationships should be seen as, for example, an appropriate mental model / schema *affects* the ability to retrieve knowledge. How and why this effect occurs is not speculated on in the diagram. All three explanatory models are student-centered in that a label refers to the student unless otherwise commented; for example, one should re-read the previous relationship as “an appropriate mental model / schema within the student effects the student’s ability to retrieve knowledge.”

These diagrams are similar to, but not the same as, causal loop diagrams. The symbology has been adjusted for the sake of clarity for the reader. In a standard causal loop diagram, a “+” (plus sign) or a “-“ (minus sign) indicates if the relationship is reinforcing or retarding; my experience suggests that readers can not see the minus sign and thus this semantic aspect of the diagram is lost. In addition, not enough is known about these interactions to appropriately categorize them as reinforcing or retarding.

Within each section is an explanation of the elements in the diagram; they are placed in alphabetical order to make each element easier to look up. Each explanation section briefly describes the variable and gives references to specific quotations from the literature (contained in appendix A and indicated by numbers in square brackets) which generated the variable description.

Cognitive aspects

Please refer to figure 4-1. This model shows the 18 elements of the cognitive domain of LP/DE as determined by the study.

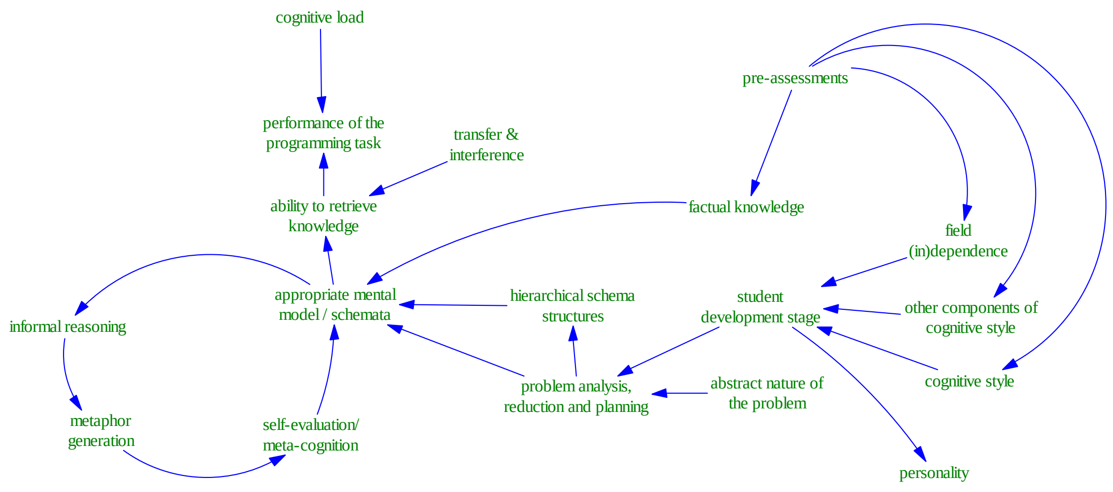
cognitive load
performance of the programming task
transfer & interference
ability to retrieve knowledge
pre-assessments
factual knowledge
field (in)dependence
informal reasoning
appropriate mental model / schemata
hierarchical schema structures
student development stage
other components of cognitive style
cognitive style
metaphor generation
self-evaluation/ meta-cognition
problem analysis, reduction and planning
abstract nature of the problem
personality

Figure 4-1: Cognitive aspects of LP/DE

Ability to retrieve knowledge

Any model of LP/DE will require separating *storing* the material from being able to *recall* it [1]. Generally we know that recall is contextual, and so the student must be able to recall the proper information in the context that it is needed. Often students know the relevant command structures but are unable to recall them in the appropriate context, or they consistently misuse an aspect of a programming language [2]. In other words, it's not that the student does not know or can not recall it the material, but that s/he has stored it in a systematically incorrect manner [3]. This makes sense from within a constructivist framework because when students construct their own meaning, sometimes it does not match the expectations of the instructor [4]. Some student responses that seem bizarre at first turn out to be reasonable when interpreted within the student's constructed framework.

Abstract nature of the problem

A programmer is not manipulating something concrete, but instead an abstract domain which must execute properly in a different time, place and context [9]. This level of abstraction is unusual in human experience. Computer programming is more abstract even than other abstract domains, including physics and mathematics, because there is no readily obvious physical analog to an executing program [5]. Programming is unlike, say, geometry, where perception of the "real world" constrains the premises [6], and more like, say poetry, where there is little limit on what can be expressed. This lack of constraints on the expressive power of a program allow for a conflict with the physical nature of the program-executing-in-the-world, causing a disconnect between the executing program and user requirements [7]. This largely unconstrained environment can cause the novice programmer to "freeze" under the weight of so many possibilities. In addition, although a programming language might resemble a natural language such as English, there is not a one-to-one semantic mapping between them [8].

The difficult part of learning to programming seems to be learning the fundamental concepts of sequence (“goto”), iteration (“do…while”) and decision (“if…then”) [10]. Once these are mastered, the student may make very quick progress with more complex constructs. However, novices often try to make their programs concrete; that is, they focus on the concrete aspects of the program and miss the more abstract nature of the process [11].

Appropriate mental models / schemata

A mental model (or schemata) is a representation of a real-world system which allows the model-holder to predict behavior of the system. A poor mental model of an executing computer program will cause the novice to write a program which does not perform as required; in addition the student will not understand why this is happening [12]. These mental models are in a constant state of flux, especially while the student is first learning [13]. An appropriate mental model is necessary in order for the programmer to appropriately abstract over circumstances [14]. Development of cognitive models and metaphors helps the learner/programmer deal with that complexity, helping to know when to apply the appropriate command structures of a programming language and to differentiate essential from non-essential aspects of a programming problem.

Further exacerbating the problem is the unnatural concept of an *internal* *state*, which is fundamental to the concept of a digital computer. This idea can be daunting for new students of programming [16]. Having a mental model which includes this notion is paramount to successfully learning to program.

As a programmer gains experience, certain situations are encountered often enough that concept “chunking” takes place, negating the need to return to the mental model of the computer [15]. This chunking may explain why many experienced programmers have a difficult time explaining the fundamentals of programming to novices.

Cognitive load

The concept of cognitive load is central to the cognitive theory of learning; the concept is that there are processing speed and space limitations in the mind of the learner, and that past a certain threshold of speed or space requirements no learning can occur [17]. Cognitive load theorists generally attempt to minimize cognitive load while either holding *transfer* constant or (ideally) increasing it. Cognitive load can work synergistically with short-term memory capacity, inexperience, stress and other factors to make a particular learning occasion ineffective.

Short-term or working memory capacity comes into play in programming because of the complexity of the semantics of most modern programming languages [18]; this may be one reason why working memory has been found to be a sufficient predictor of programming ability (Lehre, Guckenberg and Sancilio, 1988, p. 102).

The difficult task of lesson design is to make the task cognitively challenging enough so that learning occurs without making it so difficult that the student is overwhelmed and shuts down. For example, the choice of completion problems over traditional “from scratch” programming assignments has been found to be an effective way to manage cognitive load while teaching a difficult lesson [19] (however, the usefulness of completion tasks is debated; see the section on *course design*). According to Ko and Myers (2005), programming errors are caused by the interplay between cognitive load, the local environment and the programming software employed. (See also the section of *software environment*.) McKay (2000) notes a relationship between cognitive load and *cognitive style*, noting that lessons designed in opposition to the student’s learning style will increase cognitive load and thus decrease transfer.

Another factor that increases cognitive load is context switching [20], which is the amount of time required to switch between cognitive tasks. One student in this study reported the problem this way:

> *With [problems] comes asking the teacher for help, and waiting for a response. This is frustrating, because I have to move on to another subject while I wait for her to help me out. Often times I am halfway through something else when the email finally shows, and I have to drop that to finish what I had started possibly hours/days before.*

Cognitive style

Students' approached to programming seems to be influenced by their cognitive style [21]. The idea of *cognitive style* is that each person has habitual ways of remembering and thinking, and that, in the absence of other factors, these will tend to dominate. When put under pressure, a student will revert to his or her preferred cognitive style, even if it is known to be counter-productive [22]. This makes the design of instruction treatment particularly important for a high-cognitive-load activity such as programming [23]. Of the four categories of cognitive style identified by McKay (2000) (wholist-verbalizer, analytic-verbalizer, wholist-imager, analytic-imager), three of them worked best with a graphical model while the analytic-imager performed best with a textual treatment . This indicates that we can increase learning by designing multiple instructional formats to accommodate cognitive style [24] [25].

van Merrienboer (1990) investigated the use of cognitive-style targeted compensatory pedagogical approaches. After categorizing students by their reflexivity-impulsivity cognitive style, he attempted to remediate the students' difficulties by trying to force them into the opposite mode. This only led to frustration on the part of the student and, after obtaining these results, he suggests that the student be allowed to choose one or all of the types of presentations available, rather than to be assigned to one [26]. There is an aspect of this in student self-selection of course delivery; as one student in this study who preferred LP/F2F courses said:

Computer science [by] distance learning gets annoying after awhile, maybe because I like to hear and see things rather than just read.

But the instructor is not always able to make specific accommodations for each student, especially in large classes [27]. The transactional distance (Moore and Kearsley, 1996) created either by geography, temporality or crowding can make individual assessment of optimal learning treatments based on cognitive style daunting.

Factual knowledge

According to Bloom's taxonomy, the first stage of learning is a basic mastery of facts; this must precede the development of any further learning. And so it is for programming; there is a core set of basic facts which must be learned by the student in order for programming skills to be able to develop [28]. There is debate over whether to teach programming by means of elaborating details on how computers work or to focus mostly on programming as a linguistic activity (see *course design*). However, in either case students will still be required to retain some basic facts; even “open book” exams require the student to know enough about the subject to know where in the book to look (see also *assessments*).

Field (in)dependence

There is evidence that field independent learners work better in computer-assisted environments as compared to field dependent students [29]. Field independent people have the ability to restructure information to suit the needs of the individual [30], while field-dependent people tend to conform to the way that information is presented to them. Field-dependent learners prefer well-structured material (Parkinson, Redmond and Walsh, 2004) and may benefit from the use of a “map” of the material.

The ability to learn to program is highly correlated with field independence (see, for example, Mancy & Reid, 2004 and Bishop-Clark, 1995) because designing a computer program involves manipulating the essence and structure of the data available in order to solve the problem at hand [31]; the correlation is even stronger between field independence and design than it is between field independence and coding (Bishop-Clark, 1995). In addition, Foreman (1990) found significant correlations between field independence and program comprehension (r=.35), program composition (r=.38), debugging (r=.47) and program modification (r=.37).

Hierarchical schema structures

Davies (1994) proposes that computer programs are stored an manipulated hierarchically in the mind of the proficient programmer, where the central concepts of the program (what Davies calls *focal lines*) occur higher in the organizational structure [32]. Pirolli (1993) proposes that solved programming problems are stored in a tree structure which gets filled in with experience; the novice programmer accesses the decision tree looking for the best model to use. This is similar to the cognitive psychology / artificial intelligence concept of *frames* (Minsky, 2000). However, since each student develops his or her own internal organizational structure, idiosyncratic notions about programming often occur [33]. Such conceptual structures can be purposely developed within the student through appropriate instruction; making idiosyncratic (i.e., wrong) views of programming less likely [34].

In programming parlance, *structured programming* is the activity of breaking a large program into smaller constituent parts, each of which may themselves be further broken down. Foreman (1990) believes that this approach to programming works because it lessens the cognitive load on the student [35]. (See also *cognitive load*.)

Informal reasoning

All of programming is based on a single logical deductive operator, the *nand* (not-and). However, although all programs can be described at some level as a sequence of nands (much as a mudslide could be described as an interaction of sub-atomic particles), usually the student works at a higher level of abstraction than this. And, since a computer language is a subset of the domain of all languages, it is not possible to teach programming by teaching the student every possible program s/he will ever write. This requires the student to make certain leaps of thought and therefore requires informal, non-deductive reasoning [36].

Thus, research indicates that the ability to reason informally (especially inductively) is a significant factor in a student's ability to write programs [37][38].

How best to instigate appropriate inductive thinking is still an unanswered problem in LP/DE. This is a place for further research, with the goal of introducing programming concepts in a way which supports the development of appropriate inductions [39]. For example, it is possible that completion problems are better than "from scratch" problems for developing students' cognitive approaches to programming [40].

Metaphor generation

Due to the *abstract nature of programming*, including designing problems based on metaphors (models) of the real world, it can be argued that metaphor is a fundamental aspect of LP/DE [41] and that the ability to develop models of situations is essential to learning to program [42]. This is no mere neat cognitive formalism; some argue better programming can be taught be explicit instruction in program design strategies, including real-time "messy" demonstrations of programs being designed and built, as opposed only to seeing the results as a finished final product [43]. This might be implemented in a F2F course by having the instructor solve a programming problem "from scratch" while the class looks on via an overhead projector. In a DE environment, this might be done via video or by supplying the student with an animation (including verbal commentary) of the instructor solving such a problem. For example, Macromedia's *Captivate* allows the user to capture every key click and mouse movement so that the session can be played back at a later time. (http://www.macromedia.com/software/captivate/productinfo/overview/).

Poor choice of metaphor can be damaging. Novice programmers sometimes treat the computer as though it can think on its own and/or “know what I mean” [44]; the inability of the computer to do this causes the student to focus on the syntax of the commands in contrast to the semantics of the program logic [45]. (See also *problem analysis, reduction and planning.*)

There is a movement in software development called *design patterns* which seeks to develop meta-level solutions to commonly occurring problems which the programmer can then use to solve a particular programming task (Gamma, Helm, Johnson and Vlissides, 1995) .   This might help novice student to develop their ability to solve more complex problems [46], my own analysis of some of the standard design patterns indicates that they are too abstract to be useful to the beginning student.

Other components of cognitive style

The primary cognitive style measure found in the research is the *field dependence/ field independence* continuum. However, there are other measures of cognitive style. For example, it is widely believed that the Myers-Briggs type indicator is well correlated with the choice of software development as a profession; however, there does not seem to be evidence that the Myers-Briggs type represents a good predictor of *ability* in programming [47]. Another cognitive style is "fluid" ability (Foreman, 1990), which relies less on previous experience and more on an ability to learn-as-you-go [48]. Direction-following, while not precisely a cognitive style, has also been measured to highly correlate (r >= .58) with programming ability [49]. The good news here is that direction-following is a teachable skill, and thus deficiencies in this ability can be ameliorated.

Mismatches between the student's cognitive style and that of the instructor can cause an increase in cognitive load. However, Bruce (2004) finds most computer science instructors are abstract-first learners, whereas most introductory students are not [50][51]. She conjectures that this difference might make the intuition of many faculty a poor benchmark of pedagogical approach.

Performance of the programming task

Modeling how exactly a person performs the programming task is the subject of much of the research published by Davies (e.g., 1991 and 1994) and Mayer (e.g., 1985 and 1988), who attempted to work out a detailed cognitive explanation of the phenomenon. Programming is hard for several reasons, such as the abstract nature of the problem; however, one aspect that is often overlooked in the literature is the *pickiness* of a programming language. Often students are not accustomed to being subjected to a rigorous analysis of their work. Even in math classes, the instructor is likely to allow the student to skip intermediate steps; in programming, this will not work. As one student wrote:

> *Another problem I've had is spelling errors in Java. They are very annoying, and unless it's written perfect, you often get compilation errors.*

One common issue found in the literature on LP/DE is that of creating the proper balance between expository versus task-based knowledge-building. Some (e.g., Madison and Gifford, 2002) claim that instruction is typically too task-based [52]. Months or even years into programming, students still demonstrate conceptual misunderstandings about basic aspects of the field (McGill and Hobbs, 1996 , p. 73). This issue highlights an important unresolved question in LP/DE: should introductory students be taught how computers work at a fundamental level, or should introductory courses be taught using the programming language as an abstract description language which is unrelated to any physical manifestation of machinery? (See also the section on *course design*.)

Personality

Personality traits, along with demographics, are perhaps the most studied aspect of LP/DE students. However, the results are mixed. Bishop-Clark (1995) claims that "although many personality traits have been studied..., few have been shown to relate consistently with achievement" (p. 373).

Prasad and Fielden (2002) discuss the use of the Myers-Briggs Type

Indicator in programming domains, indicating that most people expect MBTI "thinkers" are likely to perform best at programming. However, empirical research calls this assumption into question, possibly because programming is not a monolithic activity and perhaps because different types do better on different aspects of programming. They specifically note that "for the most part, cognitive styles and personality traits have failed to consistently explain individual differences in programming" (p. 374). They conclude that one of the reasons that for the inconsistency is that programming achievement is variously measured by either the ability to code or the ability to design, and that these require different approaches to thinking.

Personality traits are most properly seen as inclinations not limitations (see also the section on *cognitive style*). It is quite possible that some students will *enjoy* programming more than others, although it does not follow that they will necessarily be better at it, as indicated by Tsai (1992).

Some LP/DE instructors believe that they can tell how well a student will do based on their personality; for example, one instructor wrote:

> *Within two weeks I feel I can typically predict final grades with an accuracy of 85% +. It is a combination of past experiences and their style in the class - time management, ability to ask for help efficiently, level of frustration tolerance, sense of humor.*

There is also and aspect of circadian rhythm; some people (especially in their late teens and early twenties) have extreme difficulty functioning within the traditional “banker’s hours” that most college classes operate within.

> *I'm a late night person and I hate waking up early. If I had to wake up early for classes I would most likely skip them a lot. For online classes there is nothing to skip, which is what I love about it.*

Pre-assessments

One goal of research into LP/DE has been to develop assessments which can be given to students at the beginning of their studies in order to determine their likelihood of success. Several researchers (see, e.g., Mayer, Dyck and Vilberg, 1986 and Lehre, Guckenberg, Sancilio, 1988) found that students' ability in math word problems is a significant indicator of the ability to learn to program. A student's ability to program has been found to be significantly related to his/her scores on generic tests of translating a problem statement into a formal language (e.g., as in algebra word problems) and a test of ability to comprehend a procedure (Mayer, 1985, p. 125). McGill, Volet and Hobbs (1997) found that course completion was related to planned hours of programming work, confidence in completion and expected achievement in programming.

But knowing success factors may be of little use to the instructor, since most do not have the luxury of choosing students [54]. Without administration support for enhanced entrance criteria (which is unlikely), instructors in introductory classes must deal with the wide variety of students that present themselves in class. Given enough information, students may be able to gauge if a DE course is appropriate for them, as indicated by an instructor in this study:

> *We often see stronger students self-selecting online so they are often stronger students in this and following courses.*

Perhaps a pre-test could help the student determine if DE is an appropriate delivery method. Katz, Aronis, Allbritton, Wilson and Soffa (2003) show evidence to indicate that a relatively short (on the order of four hours) on-line tutorial could be effective for this purpose [55].

Problem analysis, reduction and planning

One student from the study wrote about a problem she encounters in programming:

> *For me, I think of how I would write the program and I end up with like three steps. In reality, it ends up being like twenty, because I have to sit down and think of the logical direction the computer takes because the computer doesn't know anything starting out. It only knows what you put in that particular program. It has no experience, obviously, as a person would. Sometimes I get frustrated in trying to think of how it should be written and it seems so pointless to write it in that fashion. I have realized over time, the importance of writing the programs this way. Especially when you want that same program to do different things later on. It must be very detailed.*

A standard approach to complex problem solving is that of breaking a large problem into smaller, more manageable component pieces. This is very common in computer programming (it's called top-down refinement); although it does not always work for sufficiently difficult problems, it is the approach of first resort for most novice-to-intermediate programming tasks.

However, this type of problem analysis and reduction requires some complex cognitive functioning [56], and not every student is able to do this (see *student development stage*). Sometimes students continue to use trial-and-error methods and never break into the design and planning stage. For example, one student in this study described his programming sessions as follows:

> *Whenever I would ask the teacher for help she would point to the error messages, something I could also see on my own too, and would tell me to read the book, after reading the book I still would be kind of confused and would mess around with the program, listening to the compiler or adding a semicolon here or taking one out there which would complicate and screw up the program up even more. Sometimes it took me hours even weeks just to write a simple program.*

In addition, there is a problem with the notion of top-down refinement, in that it is optimistic to believe that one can take an unfamiliar problem, break it down into smaller unfamiliar problems, and somehow obtain the expected solution (Jackson, 2003). There is disagreement as to how programming experts approach the task; some claim that experts spend a lot of time designing and planning [57], whereas many experienced programmers claim to simply sit down and start writing. This disagreement may rest on the definition of what an “expert” programmer is; some researchers count a programmer with four years of experience as an expert (Pfleeger and Atlee, 2005), whereas life-long programmers may find a person with that amount of experience to be barely in the category of “professional.” And, of course, not all experience is equally valuable, and so a simple year-count is almost assuredly inappropriate.

How an expert programmer actually performs his or her work may be as individual as a learning style or choice of work environment; there may be no one single definition of how an expert programmer performs the task. Whereas an “assembly line” approach has been suggested by some (see Pfleeger and Atlee, 2005), some (e.g., McBreen, 2001) have called for a return to a craftsmanship model wherein individual stylistic differences are celebrated instead of trying to be removed.

Self-evaluation / meta-cognition

describe student learning as a continuous self-evaluation loop, where the student watches the expert while the expert scaffolds the student; this works as long as the expert is able to present the domain in a way which challenges but does not overwhelm the student [58]. (See also the sections on *scaffolding* and *interaction with the instructor*.) Ideally, a programmer will also continue to monitor the effects of his or her current work on the work that has come before (Lehre, Guckenberg, Sancilio, 1988, p. 85), which is another way that programming taxes cognitive resources. This self-evaluation loop requires mindfulness [59] [60].

But Murphy and Tenenberg (2005) question whether students “know what they know” and Lahtinen, Ala-Mutka, Jarvinen (2005) note that students very often overestimate their own level of understanding. Murphy and Tenenberg (2005) found that the lowest-performing computer science students were the most likely to make inaccurate predictions regarding their own level of knowledge. This means that relying on students to ask what elements of the course to review (for example) is probably a poor strategy and instead the instructor should provide ample opportunity for objective assessment (for example, automated quizzes, etc.).

<u>Student development stage</u>

According to Arzarello, Chiappini, Lemut, Marara, and Pellery (1993), an important part of learning to program is the “sudden transition from simple computational attempts to the synthesis of the whole in a procedure or program” (p. 289). However, a significant percentage of students may not be able to make this leap, as exemplified by the bimodal distribution of grades in LP classes, where some students seem to “get it” while others “just don’t get it” [62]. An instructor in this study had this to say during the third week of class:

> *Yes, there are differences between students this early in the "game". The students who have extreme difficulty in programming (called non-programmers) struggle with understanding even simple ideas such as variables and why you need them and how to get rid of simple compiler messages. The average student is struggling with the logic*

of beginning programming. The good students (called programmers) see all the concepts, apply them well and need little help because they "get it".

This was echoed by another instructor at the same time who wrote:

I can also identify two broad categories – those who get it and those who don't.

One of those students in the study who was not a programmer and perhaps was not “getting it” describes the situation as follows:

The class is much harder than I thought it would be. I don't believe that any course at the 100 level should be this difficult. I am a relatively intelligent person but I am not a programmer. The instructor, I believe, assumes that all of the students are programmers, or perhaps is just out of touch with the reality of the challenge of learning something like this when you have little programming background… This is the "pits.”

Applying Piagetian theory may supply an answer to this conundrum, specifically involving concrete versus formal operations stages of development [62]. Analysis of computer science students with respect to formal operational development resulted in 72% overall correct categorization into successful/ unsuccessful categories, leading to the conclusion that novice programming students should be evaluated along these lines before signing up for class [63] [64].

But programming is not a monolithic activity; it requires various kinds of abilities for different aspects of the domain. For example, programming requires attention to detail; one student in the study described his situation as follows:

> *I'm a very abstract learner. I see the big picture but its ONE or two tiny little details that are crucial that put me behind. She seemed so mad that I would get datatypes and constructors mixed up, it was all new to me -- it was going to take time and being a teacher she should have had some patience with me. She did not take the time to help me find an alternative way to learn the material.*

Program comprehension – being able to understand a program that someone else has written – is well correlated to logical reasoning ability (r=.67) (Foreman, 1990) and therefore to the formal operations stage. Program design differs from program comprehension and program construction, and Buck and Stucki (2000) suggest that many introductory students are exposed to design too early in their tutelage.

Transfer & interference

In addition to obtaining factual knowledge of, and task ability in, a specific domain, it is desirable for a student to be able to *transfer* knowledge from one domain to another. Transfer generally means that what is learned affects future performance, and can be of many types, for example, transfer between tasks or between contexts. van Merrienboer, Schuurman, de Croock, and Paas (2001) found that using completion-type problems (versus traditional “from scratch” problems) resulted in better near, intermediate and far transfer. They attribute this to the nature of completion problems as decreasing extraneous cognitive load, thus perhaps allowing students to spend more mental energy encoding the information into their cognitive schemata in a more dynamic way.

However, the problem of negative transfer or an "einstellung effect" may result in a student applying a less-than-optimal solution simply because the student is familiar with that solution or because such a solution was recently posed (see, e.g., Arzarello, Chiappini, Lemut, Marara, and Pellery, 1993).

In a loose manner, the opposite of transfer is *interference*. Two types of interference are discussed in the LP/DE literature: proactive and contextual interference. *Proactive interference* occurs when a student uses the wrong schema in attempting to solve a problem [66]. For example, by mixing up the semantics of a “while” command in C++ with the English-language meaning of the term [67]. Another example of proactive interference within the context of this research is a student’s previous experience with programming; surprisingly, students who have learned one programming language may have a more difficult time learning another than those with no experience at all [68] (see also the section on *previous programming experience*).

In contrast, *contextual interference* is a pedagogical tool used to increase student learning by increasing germane cognitive load (van Merrienboer, Schuurman, de Croock, and Paas, 2001 p. 12). By forcing the student to contrast the new information with some other, already known, domain. This result could manifest itself in teaching LP/DE students by requiring them to compare and contrast the target language with other languages that they are familiar with – for example, English or a pseudo-English intermediate language. van Merrienboer et al found that high contextual interference is inversely proportional to conventional (from “scratch”) problem solving when efficiency is held constant. Thus, the worst educational situation was the high contextual interference situation with from-scratch programming problems. The highest training efficiency was in a high contextual interference situation with completion problems.

The research seems conclusive that transfer and interference is an important aspect of LP/DE, and that instructors must be mindful of these issues when designing curricula and materials.

Pedagogical aspects

Please refer to diagram 4-2, which should be analyzed in the same manner as diagram 4-1.   Pedagogical aspects of LP/DE involve those elements primarily under the control of the instructor, but which affect the student, who can in turn affect other students and also the instructor. There are twenty-one elements in this part of the model.

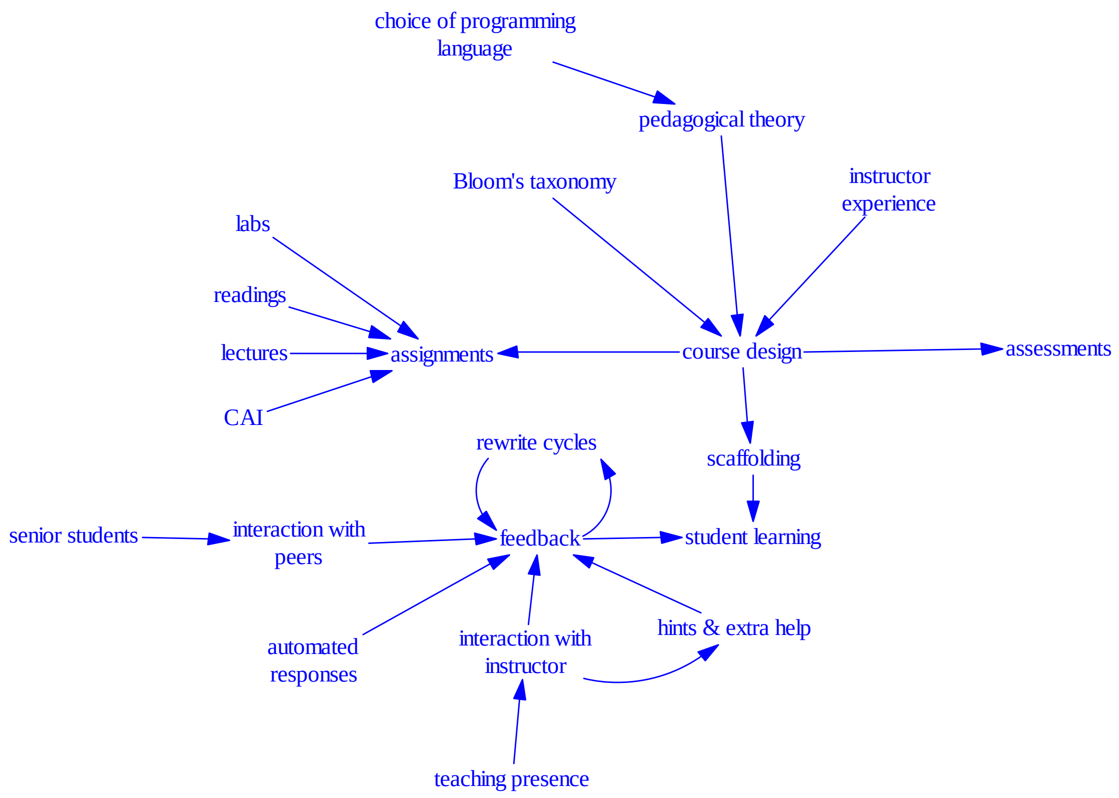

Figure 4-2: Pedagogical aspects of LP/DE

Assessments

Many theorists, especially those from the school of instructional design, believe that assessments are an integral part teaching [70], however instructors find grading of DE courses to be more cumbersome that F2F courses (Kleinman & Entin, 2002). With regard to assessing a student program, there are classically three aspects: verification (does it do the job right?), validation (does it do the right job?) and standardization (does the style of the program conform to coding standards?). Different instructors will weigh these aspects of a program differently; for example, some will not consciously differentiate them, and some may ignore style almost entirely (i.e., “if it works, it's good”). McCraken et al (2001) discuss the two most common types of assessment in LP, which are take-home programming assignments and multiple-choice tests. They note that the former tend to be fairly large-scale, generalizable, are ecologically valid, and tend to have a generous amount of time allotted for their completion; however, they penalize students who are unable or unwilling to spend the amount of time necessary to complete them. Multiple choice tests, though often used, are thought by some to be neither meaningful nor generalizable [71].

Woit and Mason (2003) studied four ways of combining assessments for purposes of course grading: (a) partially marked laboratories, no midterm; (b) voluntary laboratories with online midterm; (c) marked laboratories with online midterm; (d) online weekly laboratory quizzes, no midterm. All study groups had an on-line final exam. Category (a) resulted in very poor performance on the final exam, and much sharing of graded lab assignments was reported. Group (b) was the next poorest performer on the final; many students reported not doing the lab assignments at all, which left them very poorly prepared for the exams. Group (c) did somewhat better in the final, although cheating still occurred, especially among those students who ended up doing poorly. Group (d), with an online weekly quiz, no midterm, and no “points” for doing the labs did the best on the final exam. This concurs with Shaffer (2005), who structured an introductory

F2F LP course in the same way; in both studies the weekly quiz material was taken from the lab assignments. Woit and Mason also conclude that students routinely do poorly on their first on-line programming exam, and thus it is important to give the student ample opportunities with this kind of assessment before foisting it on them for a final exam. They further find that students agree that the on-line assessments are a better indicator of programming ability than the traditional paper-and-pencil / multiple choice exam, and others agree [72].

McCraken, Almstrum, Diaz, Guzdial, Hagan, Kolikant, Laxer, Thomas, Utting and Wilusz, 2001, p. 127); introduce another type of assessment, which they refer to as a *charette*. A charette is a “short assignment, typically carried out during a fixed-length laboratory session that occurs on a regular basis” (p. 127). Because the charette is proctored and timed, *cheating* is diminished. Although this approach enables the instructor to assess the student's ability to program, by their nature these problems can not be as complex as the take-home assignments. In addition, some students who are prone to test *anxiety* may be at a disadvantage.

Carbone and Sheard (2002) discuss an additional, non-traditional, option for assessment in LP: portfolio assessment. Portfolio assessment, a common approach to assessment in the arts, consists of a critical assessment of a collection of work performed by the student. Some items are required by the instructor, some are works selected by the student. This method of assessment is of interest in LP/DE, especially in organizational models where course credit is earned in unconventional ways (see, for example, Thomas Edison State College's *prior learning assessment* approach at http://www.tesc.edu).

Assignments

While some see the assignment as the primary factor in DE [73], this seems to be seen to be at odds with Anderson's (2004b) notion of *teaching presence* as the critical aspect of online teaching. Whether central or not, virtually all courses require some sort of work be performed by the student and usually this includes some sort of review process by the instructor (see also the section on *assessments*). Whether graded or not, an almost universal assumption of any course is that the student must indeed *do something* in order to change his or her state of being from, for example, novice to intermediate. Typical LP/DE assignments include reading, doing research, creating a program or testing a program.

Automated responses

Accurate and timely feedback on assessments and assignments is almost assuredly a key factor in LPDE [74], however this is notoriously difficult in LP classes [75]. This problem can be ameliorated through the use of automated tools [76]. An instructor, when asked to describe an ideal LP/DE software environment, wished for:

"A system that grades the programs and gives the student feedback about their program by telling the students where the problems occurred and giving them suggestions for more efficient code."

For example, Shaffer (2005) describes a system for doing a detailed assessment of submitted programs, including testing the execution against an instructor-defined data set and checking for stylistic attributes. Zeller (2000) includes the ability to automatically assign programs to fellow students for peer review. Woit and Mason (2003) do automated evaluation of programs but do not grade on elements such as style, algorithms, or documentation.

Automated assessments also offer the opportunity for automated plagiarism detection, *cheating* being historically a problem in programming classes, especially large ones (for a complete treatment of this subject, see Culwin, MacLeod and Lancaster, 2001). Computer programs are now available which can detect cases where a substantial part of program was copied from one student to the next, even if surface aspects such as variable and function names are changed.

On the negative side of this issue, Thomas (2002) reports the use of an automated programming "coach" in a LP/DE environment [77]. While this sounds like a good idea, students who used the coach were found to be much less likely to read the discussion pages, and these students were found to be weaker programmers during a posttest.

Bloom's taxonomy

Bloom’s taxonomy is often used in discussions of LP/DE (see, for example, Roussev and Rousseva, 2004); it incorporates an series of learning stages that a student goes through in developing mastery of a domain. These stages are factual knowledge, comprehension, application, synthesis and, finally, evaluation. The last stage requires a mastery of the other levels and the ability to apply expert judgment and multiple solutions to a problem domain Roussev and Rousseva, 2004). Buck & Stucki (2000) point out that the traditional way of teaching programming follows the traditional software development process which is essentially the *reverse* of Bloom's taxonomy. This can work for some students, but leaves many with difficulty grasping the material [78]. (See also the section on *student development stage*.)

CAI (computer aided instruction)

Throughout the last thirty years there have been a number of attempts to create software to support LP, initially on time-shared computers and then with CD-ROM based programs. More recently, several researchers have created web-based systems for LP which may be able to be adapted for use in LP/DE (see, for example, Truong, Bancroft and Roe, 2005 and Shaffer, 2005).

Intelligent program tutors that use artificial intelligence techniques have been in existence at least since 1986 (Anderson and Skwareki, 1986). These systems can aid learning by keeping track of students' learning styles, history, prior knowledge, progress, etc. (Pillay, 2003, p. 79), and some have found that students performed better with an automated tutor than without it, but not as well as with a human tutor (Anderson and Skwarecki, 1986, p. 846). In addition, an advantage of intelligent tutors is that they may allow for detailed data collection on how a student learns [79]. Although intelligent programming tutors have been demonstrated to be effective, they have not been widely used because of high development costs and lack of shareable resources (Pillay, 2003, p. 78).

Pillay (2003) identifies six necessary features of interactive programming tutors [80]. However there is a wide range of approaches taken. Some CAIs work using a fill-in-the-blank approach, requiring the student to only fill in key sections of a program template (Odekirk-Hash and Zachary, 2001). This makes diagnosing student errors easier to program into the system, but there is a major issue as to what the student is actually learning in a situation like this; it seems that there is a big difference between writing a program “from scratch” and filling in some blanks in an existing program.

CAIs which allow more programming-like behavior and yet are still intrusive include those that monitor each student keystroke and “pop up” when the student goes off course. For example, Anderson and Skwareki (1986) present a system in which responds to each student keystroke [81]. Still less intrusive are systems that fill in the programming details for a student as s/he types. There have been a number of both professional and student-based IDEs developed which do this; however, anecdotally, expert programmers find these to be counter-productive.

Next on the continuum away from intrusiveness are those systems that allow the student to click a button for tutorial help when s/he runs into a problem. For example, Zachary and Jensen (2003) describe such a system, but this feature is only available in tutorial mode, not while doing assignments or exams. Allowing students to make mistakes is thought by many to be fundamental to learning to program.

Short of leaving the student programmer totally to his/her own devices, the least intrusive approach is described in Shaffer (2005). In this system, students work on a problem and, when they believe they are ready for some feedback, submit the intermediate version for analysis. There are no negative consequences for submitting often (in fact it is encouraged); this allows students time to think and re-think an aspect of their program, giving them the chance to recognize their errors before the system tells them what they are.

At the far opposite end of the “intrusion scale” is requiring students to learn and use a professional software development interface. But technology hurdles can be a major factor in drop-out rates (Kleinman & Entine, 2002), and the complexity of shifting through the myriad options in a professional IDE is likely to increase a student's frustration.

CAI systems that support aspects of LP/DE other than program development include simulation and explanation systems. Simulation systems primarily work by visually representing how a working program changes the internal state of the computer; although a number of these have been developed over the years, they have tended to be ad-hoc. Heo (2003) describes an explanation system of programming code annotation that allows the student to access detailed descriptions of aspects of example programs by “hovering” with the mouse over that part of the code. This approach is similar in many ways to the interactive model of LP/DE presented in the electronic version of the current paper. Heo found that “it is clear that the use of code examples with embedded description statements enhanced student learning” (p. 154).

The number and variety of CAI systems for LP and LP/DE is a very interesting aspect of the field. It seems that every researcher in this area has decided to start from scratch (including, abashedly, Shaffer, 2005). This subject by far yields the most “hits” in a literature search and yet, unfortunately, there is very little to be gleaned from this research since it tends to be of the “one-off” variety. Hope is not lost, however; the Advanced Distributed Learning / SCORM initiative of the US government (http://www.adlnet.org/) is a protocol for sharing “learning objects” with a common structure so that content can be edited and reused easily. Due to size and complexity limitations, this aspect of LP/DE was eliminated from detailed analysis within the current project, and could be the subject of an entire dissertation itself.

Choice of programming language

Although it has been accepted for some time that programming language selection will influence programmer behavior (Davies, 1994, p. 706), the choice of an appropriate introductory programming language is hotly debated among computer science instructors (see Bruce, 2004 for a review of the debate). Even those who are against the use of “toy” programming languages, and who teach complex commercial programming languages in a first course, often create a subset of the language in order to restrict the complexity of the learning environment (Bruce, 2004, p. 32). Some authors strongly advocate languages created specifically for teaching, but McCraken, Almstrum, Diaz, Guzdial, Hagan, Kolikant, Laxer, Thomas, Utting and Wilusz (2001) found suggestive evidence that what looked on the surface like a learning effect based on choice of programming language probably was not due entirely to this difference.

Zachary and Jenson (2003) base their LP/DE course on HTML and Javascript, both of which are available to any student who has access to a web browser. This helps to balance the practical needs with the pedagogical; however, they found it necessary to restrict their material to a subset of these languages due to their complexity.

With respect to the learning of future languages, Holden and Weeden (2004) found that, although prior experience was an indicator of success in an introductory programming course, no one programming language was any more likely than another to predict future success [82]. However, Morrison and Newman (2001) found differently (see *transfer and interference*).

There is more than pedagogy involved in the selection of programming languages, including market trends [83]. Many still argue that the choice of programming language one learns first is not important [84], although this point can be debated from the standpoint of the linguistic relativity hypothesis (Shaffer, 1997).

Course design

One major decision that an LP/DE instructor has to make is how much to scaffold the students in the execution of the course. One trade-off seems to be between minimizing cognitive load versus maximizing ecological validity; for example, van Merrienboer, J., Schuurman, J., de Croock, M. and Paas, F. (2001) found that completion problems (which lower cognitive load) are better than solve-from-scratch problems, even though they are less ecologically valid. Completion problems provide a bridge between worked examples and conventional problems. In an earlier paper, van Merrienboer (1990) went so far as to state that "the completion strategy should be preferred over the generation strategy in practical teaching situations" (np). Similarly, some advocate giving students a series of small problems to scaffold their understanding of programming [85]. However, the ecological validity argument states that computer programming consists of solving whole problems, not partial ones, and that therefore completion and "toy" problems do not teach the appropriate skills.

Another issue of course design is balancing student workload. For example, a student notes the following organizational problem (See also the section on time commitment):

The workload is very uneven from week to week. This makes planning very difficult. Planning is very, very important for part time students with multiple responsibilities.

This may argue for the adoption of LP/DE courses that are not constrained by semester calendars. Although difficult from an administrative standpoint, this approach must not be dismissed out-of-hand. One student in the study put it this way:

I am a big advocate for distance learning for computer programming to have sequential flexible deadlines for assignments. Some assignments take longer than others simply because you get things faster than other things. Students can pace themselves easier so the workload is not so uneven. I would think that would make grading assignments by instructors much easier too. The teamwork style of learning promoted by this instructor does not appear work from my perspective.

Feedback

Fast, accurate and personalized feedback has been identified as a significant factor in student learning (Heaney & Daly, 2004; see also the section on *assessment*). The importance of feedback is highlighted by the following statement from a student:

It is a real guessing game as to what grade I could receive now. No assignments have been graded since I last responded to survey questions. I have been waiting for over three weeks for one assignment to be graded and over two weeks for another assignment to be graded.

Even so, another assignment has been given and will be due in five days. I am totally amazed at the incredible lack of feedback from this instructor. I have begun to wonder to whom do I complain? Who monitors the actions of the instructors at the community college? Do students have any recourse? As a consumer of education, I think I should be able to demand a partial refund of tuition because the instructor failed to perform expected duties. I am not a happy student.

Often feedback for student programs is cumbersome for instructors and too superficial and late to help students [86]. One approach to offsetting this cumbersomeness used by Heaney & Daly (2004) and others is to use a whiteboard/tablet PC application to allow feedback to be written directly on the student's submitted work. This allows the reviewer to draw circles, arrows, etc. to indicate how the work product could have been improved. The present author is currently participating in a study of such an approach (http://css.its.psu.edu/news/nlsp05/notordinaryclassroom.html).

Shaffer (2004) describes the Ludwig programming tutoring and assessment tool, which presents automated feedback as students work on their programming tasks, including stylistic comments embedded right into the student programs.

Another way that timely feedback can be generated is through pair programming tasks, where students develop program solutions in pairs, ideally working with a variety of classmates during the course (*see interaction with peers*). While pair programming has been shown to decrease questions asked of instructors and assistants, Heaney & Daly (2004) point out that it has not been conclusively shown to increase student learning. They go on to suggest using second-year students for such feedback, which echoes work done in software quality assurance wherein more experienced programmers review and discuss program code with junior level employees. (See also the section on *senior students*).

Sometimes students can present answers that, although perfectly reasonable within their constructed framework, are almost incomprehensively wrong (see the section on *appropriate mental model / schemata*). This being the case, instructors should be careful not to too quickly decide that the student is lazy or incompetent [87] and give the student the opportunity to explain what s/he had in mind when submitting the work. Madison and Gifford (2002) give examples of student misconceptions which did not initially present themselves in erroneous student programs, but laid dormant until later work revealed the misunderstanding.

Hints and extra help

Since programming is a task that takes a high degree of induction and abstraction, student will sometimes get "stuck." For example, the student may simply not be able to *transfer* an appropriate model situation into the current one. Instructor opinions abound as to whether or not to help the student through this stage by giving hints, yet evidence seems to indicate that this *must* be done [88]. Well-placed tips and hints designed to build the scaffolding of knowledge are an important aspect of the instructor-student relationship (see *scaffolding*).

For those students who are struggling, Katz, Aronis, Allbritton, Wilson and Soffa (2003) found evidence to support that “exercises that promote coding accuracy and proficiency, and encourage experimentation and reflection on the results of experiments” (p. 161) are effective in helping the student move ahead.

Instructor experience

The experience of the instructor is seen by some to be a fundamental driver of the quality of an online course [89]. At least one of the instructor-subjects in this study clearly meets a high standard of experience:

> *In the early sixties, I took an educational seminar … In 1965-66, I studied for [a Master’s] in the teaching of mathematics. In the late sixties, I studied "how students learn" at ________. In 1974, I completed an Ed.D. at _________ on the teaching of mathematics and computer science … In all honesty, I learned the most about teaching methods in my first year at _______ through mentors and daily evaluation of my teaching.*

Unfortunately, this ideal is not always realized, as another instructor wrote:

> *Today, many instructors can get a Blackboard cartridge with all content, tests, web pages completely prepared for them. So now anyone can teach online and thus they grab the online classes because they feel it is "easy" and don't have to do much. It is sad.....*

In addition to teaching skills, the effective online instructor must have the requisite technical skills; experience shows that LP/DE courses are likely to draw at least some student technophiles and that these students may challenge the instructor's knowledge of certain (possible arcane) aspects of computer technology. It is important not only for the instructor to *be* competent but also to *feel* competent, thus giving the instructor the confidence to admit to not knowing everything.

Although most college-level instructors are not required to take any courses in teaching, some institutions have required them to take coursework in DE. One instructor responded:

> *In the early years of teaching on the Internet, all online teachers at our college were required to take so many seminars in online teaching. These courses were offered via the DE department and they would pay consultants to come in and teach seminars to ALL DE teachers. We were required to attend. Those requirements have also become lax in the past 2 years since the union has fought to allow ALL teachers to be able to teach any DE class without any conditions.*

Having experienced LP/DE from the standpoint of the student can be a asset in preparing to teach, as another instructor points out:

> *I took an online programming class from another college to see how others were doing the same thing as I was. I was pleasantly surprised to know that I was offering more to my students than a highly respected colleague at another university.*

Many institutions survey the students at the end of a course to obtain feedback about the instructor. Although it can be argued that these evaluations are usually skewed toward the negative, they can give valuable feedback to the instructor with respect to his or her teaching methods. Those teachers who are interested in a continuous improvement model would collect such information whether or not it's required, as the following statement from an instructor indicates:

> *Student surveys are taken for the first 4 years of teaching. After 4 years, tenure is given and student surveys then become voluntary. I do give my own student surveys and yes, I do adjust my methods and materials based upon results of the student surveys.*

Interaction with the instructor

Interaction with the instructor is key to any instructor-led learning experience, and it is an especially important focus of distance education design due to the issue of transactional distance (Moore and Kearsley, 2004). Students may get the impression that the instructor is uninterested in the success of the student, as exemplified by the following from a student:

> *The instructor was helpful when I emailed her having a problem, but I wouldn't use the word interested to describe our relationship. No, I never contacted the instructor by phone on in person, although I asked if she had any version of AIM/MSN messenger/Yahoo, to which she replied that the CCAC email service would be sufficient. I don't think the instructor would have liked me calling or meeting with her.*

According to Anderson (2004b), student-teacher interaction is an important part of teacher presence [90]; a lab-based approach to turns the role of the instructor to that of tutor [91], and on the far end of the student-teacher interaction continuum is teaching programming as a *cognitive apprenticeship* [92]. With regard to interacting with the students at a distance, synchronous solutions including computer video conferencing has been successfully implemented [93]. Where synchronicity is not possible, animations can help (see also the section on *metaphor generation*).

Some see the role of the instructor as that of a project manager, whose task is to implement a well-designed online course (Moore and Kearsley, 2004). An instructor-participant puts it this way:

> *The instructor is a manager to help the students achieve the goals of both reading and writing a program… The instructor should give immediate feedback of the student's work so the student can proceed with the next programming problem. Finally, the instructor should be able to bring together all of the material so the students can produce a final project. This will allow them to see how the information is utilized in the programming industry.*

But this approach may be seen by the student as eliminating a critical aspect of the course:

> *Difficulties unique to this class probably have more to do with the online format rather than the material itself. The online format really takes away what I view as the most [important] ingredient to learning, which is*

instruction. This class has virtually no instruction aside from the student learning and following the text.

Not every student is interested in developing a relationship with the instructor, however. For example, one older student, when asked about his relationship with his LP/DE instructor, responded:

Pleaseeeeeeeeeeeeeeeee, I do not have a relationship. I do not want a relationship. At my age I do not want or need a mentor.

At the opposite end of the spectrum, another student commented that:

I don't know the instructor well. I have only emailed her a few times asking for help/advice on programs. I sometimes get the feeling she thinks I'm stupid for asking all my questions. I definitely have a better relationship with all my other [F2F] teachers.

One common theme with students in the study was the long turn-around time of answers from the instructor. One solution often put forth by students was to have firm virtual office hours.

<u>Interaction with peers</u>

The value of peer-to-peer interaction in distance education is well-known (Moore & Kearsley, 2004). Fostering such interaction via the web avoids the feeling of isolation that often happens in a distance education environment and is an important aspect of student satisfaction [94]. However, forcing peer participation is debated, in that it "seems to recall the onerous practice of attendance marking that rewards the quantity and not the quality of participation" (Anderson, 2004b, p. 281); one student stated his objection to this practice quite clearly:

> *The instructor has this fantasy that we can work together through the message board as a team. I have tried to politely feed that fantasy through the message board but I am getting very tired of doing so. A team requires a leader, someone that can lead by example and explain how the team is supposed to function. Unfortunately we simply do not have a leader in our instructor and the format has been a total failure… My purpose for taking this class was to learn … programming in today's environment. I took distance learning so I could work independently. I deeply resent how the class is being run. It is affecting my desire to study and complete coursework.*

The counter argument notes that if students are not given "points" for interacting, they probably won't. However, one instructor from the study, who encourages but does not require interaction, averaged 120 student postings per week for a class of 13 students, indicating that student interaction was intrinsically valued by the students.

Ramalingam, V., LaBelle, D., Wiedenbeck, S. (2004) point to research which shows that watching a peer perform a task is an aid to learners, because it shows the student that someone "like me" can do the task. There are a number of studies which demonstrate the effectiveness of students working in pairs or groups with regard to learning to program (see, for example, Fernandez and Williamson, 2003), and group work was advocated in the four-institution collaborative teaching effort discussed in Cohen and Boisvert (2004). In a study of 63 computer science students, Anthony and Payne (2003) found that outside group collaborative work was significantly correlated with student achievement based on student exam scores. In a study of 134 students, Jehng (1997) found that students who worked on programming problems collaboratively did better on a posttest than those who solved the programs individually.

However, not all students respond positively toward peer interaction in LP/DE; for example, one student commented:

> *I am disappointed that the format of the class tries to duplicate a traditional classroom. The instructor wants to encourage teamwork... There is this very cheesy message board where we are required to post questions and answers. The turnaround time for posts is too long to make it of any use to me. I personally find the requirement to post rather silly and annoying. I find the team approach to education that is in vogue presently to be very insulting to the older learner. I do not want the classroom to try to mimic my work environment. I know how to work with people. I've been doing it for years, thank you very much.*

Another student had a more pragmatic reason for questioning peer interaction:

> *It is … hard to ask for help from other students as everyone has a different way of writing programs or they are very clueless themselves and may not always provide correct information.*

Peer-to-peer and animation technologies can assist in student interaction for

LP/DE. In a study of distributed peer interaction of LP/DE students, Kollelman, van der Mast, van Dijk and van der Veer (2000) found that students were very positive about working in virtual groups, however technical problems can be relentless and cause student frustration. Barring the availability of synchronous peer-to-peer interaction, animations of student development sessions could enhance the novice student's learning.

Peer-to-peer interaction is well-studied in the world of computer programming, where it is referred to as pair programming; there are many anecdotal reports on the effectiveness of pair programming among professional programmers (see, e.g.,Williams, Kessler, Cunningham and Jeffries, 2000); however, some of the quantitative studies performed do not confirm the anecdotal findings [95]. With regard to students, Thomas, Ratcliffe and Robertson (2003) found that students with the least programming confidence liked pair programming the most, while those students with the most programming confidence liked it the least, especially when they are paired with low-confidence students.

Clancy, Titterton, Ryan, Slotta and Linn (2003) propose another way that student groups can interact, by requiring them to come up with several different, though correct, solutions to a proposed problem, thus reinforcing both the notions that this is a “do-able” task and also that a problem may have multiple “correct” solutions.

Sitthiworachart and Joy (2004) discuss a system for student peer review of submitted programs, which is claimed to improve the quality of students' subsequent submissions. They suggest that specific criteria, graded on a Likert scale, be established; 94% of their student subjects agreed that, to be effective, peer reviews had to be anonymous. (See also the section on *feedback*.)

Labs

In face-to-face contexts, many instructors advocate a laboratory environment instead of lecture for LP courses [96]; during a lab, when the instructor notices an issue, s/he would stop the lab and lecture for a short time. Shaffer (2005) describes a similar approach, which has the added benefit of forcing each student to keep up with the rest of the class.

In an LP/DE environment, this is more complex to implement. Various methods of synchronous and asynchronous interaction with both the instructor and peers may ameliorate these problems. (See also the sections on *interaction with the instructor* and *interaction with peers*.)

Lectures

Adult education and DE literature has much to say on the subject of lectures, the standard metaphor being "the sage on the stage" versus the "guide on the side." In arguing against the latter – which he refers to as a *laissez faire approach* – Anderson (2004b) points out that "a key feature of social cognition and constructivist learning models is the participation of an adult, or expert, or more skilled peer who 'scaffolds' a novice's learning. This role of the teacher involves direct instruction that makes use of the subject matter and pedagogical expertise of the teacher" (p. 287). (See also the section on *scaffolding*.)

A well-performed lecture can help instill in the student a cognitive map of the domain, reinforcing and adding to the material learned from textbooks and other instructional material. Within the realm of LP/DE, it is possible to include video taped lectures, animated worked assignments with audio commentary, and possibly synchronous discussions using video conferencing technology.

Pedagogical theory

Although many LP and LP/DE instructors have never studied education theory, every instructor approaches teaching – consciously or unconsciously – with a pedagogical theory. Many of these are ad-hoc hodgepodges of partial theories, but some, though rough, are cohesive. For example, one instructor, when asked how student learn to program, responded:

> *I wish I knew. I think it first has to start with problem solving - thinking in logical steps. Seeing some standard solutions may help - if they can't model their programs after similar programs, then they probably don't get it. Eventually they have to go out of the box and come up with solutions of their own - I don't know when that happens. Syntax is another issue which definitely can be learned - that's pretty straightforward. Terminology is another issue when they keep changing the words that refer to the same thing - function, method, procedure, subroutine etc. That's what makes it hard for the students to read the*

text. The texts don't have enough examples and meaningful exercises to practice concepts. It's sort of like the question how do you teach students to write. Some of it is creative - thinking up a solution. But then there's the basic rules of the language that can be learned like math rules - through practice.

Another explained the process as follows:

I believe that students learn to program by first learning the syntax of the language and seeing samples of very simple programs... Once a student knows the general idea of how to set up some simple programs in a language, then logic is introduced... Once students have learned the syntax and seen a variety of programs with varying logic, they are ready to write programs from scratch that are different from anything they have seen. This is the hardest part for my students... The last portion of the course is doing advanced techniques in programming and expanding on logic. By this stage, students are feeling a bit more confident (hopefully) or have withdrawn from course due to not being able to do logic in earlier phase.

Note that these approaches are quite different, the first being analogous to a “whole language” approach in teaching reading, while the second is a grammar based approach. It is quite possible that these instructors have thought deeply about their approaches, studied the applicable data, and experimented with various approaches until they found an approach which worked. However, it is also possible that any number of LP/DE instructors use only their intuition to inform their teaching, yielding less than ideal results.

During the last ten years or so, the LP and LP/DE communities have developed communication venues, and thus there have been more collaborative efforts between computer science instructors and educators. One indication of this is the common reference to *Bloom's taxonomy* in the literature; one reason for the popularity of this model is that it might fit well within the algorithmically-minded computer scientist's mind.

Many LP and LP/DE instructors have underlying cognitive assumptions in their course design; however, published accounts of LP and LP/DE have included elements of behaviorism, andragogy, activity theory and especially constructivism. One older student in the study was unimpressed by the pedagogical assumptions of the LP/DE course he was taking:

> *The kind of class that I would prefer would be one that was totally independent study. Someone would be at my beck and call at the end of a phone, or in a chat room, or even meet face to face if I did require help. Assignments would be sequential but could be turned in at various times, so I could spend more time on certain subjects and less time on others as I needed. Assignments would be guaranteed to be graded within 72 hours. I want to make decisions on what I learn and when I learn it. I do not want to passively accept what the instructor thinks I should learn. I want input because this is my education and I am paying for it.*

Pedagogical theory is included in this model primarily as a reminder to the practitioner that s/he is operating under a (possibly unconscious) pedagogical theory and to bring to mind Socrates' dictum to “know thyself!”

Readings

Even with the advent of multimedia learning objects, most LP/DE courses will continue to include an assigned textbook at least for the near future. Whether the textbook is on-line or printed, there is almost always a required reading part of any lesson. Selection of a text must include thought given to the pedagogical assumptions of the text and whether or not they concur with those of the instructor.

For their part, students will likely perform a *return on investment* calculation with regard to reading the material. Instead of engaging with the text, many students will instead leaf through it looking for “the answers.” Of course, this does not result in a strong cognitive model of the material and the student’s knowledge is likely to be flimsy. Frequent, well-designed assessments with some small bit of “credit” have been found to be effective in motivating the student to read the material. However, in an unproctored environment (as is usually the case with LP/DE), it is hard to keep the student from responding to the assessment by executing a keyword hunt through the text instead of actually reading it. This is especially likely if the material is available on-line in a searchable format. This inclination of the student can be somewhat thwarted by the use of half-pages of text followed by one or two short multiple choice questions, requiring that the student stay involved with the material.

However, keeping with the notion that course materials are made for the student and not the other way around, we must be mindful that not every student learns the same way. Sometimes the student has a learning style which does not lend itself to the standard “read the text, do the problems” approach. A student in the study developed a working style which fit her learning style:

> *It is easier to try out the programs first, and then look at the chapter in more detail for me. I can't comprehend the chapter information without actually attempting the program to see how it works.*

It seems that this student is not “slacking” but instead has an orientation toward active versus passive learning. Forcing this student into a passive learning mode would do an injustice to her.

Re-write cycles

The current popularity of rewrite cycles in the learning of expository writing has suggested a similar move in programming. For example, Liz Adams puts student’s work through the following gauntlet: “write [the program], be peer reviewed, rewrite, be graded, rewrite again, and be graded again” (Null, Ciaraldi, Adams, Wolz, and Hailperin, 2002, p. 250). Max Heilperin describes his experiences with “mastery homework, in which each homework problem is graded on a binary scale (mastered vs. not-yet-mastered) and can be handed in as many times as it takes to reach mastery, any time over a broad period … The homework portion of the grade is simply the percentage of problems eventually mastered… Grades of ‘not yet mastered’ are invitations to come talk” (p. 250).

Both of these approaches require an increase in instructor time and thus might be infeasible for large classes without some kind of technology support (or a cadre of TAs). (See the section on *automated responses* for more on this.)

The concept of “mastery homework” raises the concept of mastery testing (Gentile and Lalley, 2003). In this approach, a student will re-take an exam until s/he obtains a mastery-level score. However, one problem with mastery testing is having a large enough bank of test questions. One student noted that:

> *The tests are easy, because I can take them as much as I want. Basically, I can just memorize them.*

(See also the sections on assessment and examinations.)

Scaffolding

“Scaffolding” is a metaphor for building an educational support mechanism in order for the student to be able to rise to the next level of achievement. The concept of scaffolding is intimately connected to the work of Vygotsky (1980) and his concept of the zone of proximal development: “It is the distance between the actual development level as determined by independent problem solving and the level of potential development as determined through problem solving under … guidance or in collaboration with more capable peers” (p. 86).

In learning to program the student is introduced to material and ideas for which s/he can not possibly have an organizational structure (Pirolli, 1993, p. 42), and it is incumbent upon the instructor to build scaffolding for the material in such a way that the student is able to incorporate it. Scaffolding the novice programmer's knowledge structures to enable the student to refine existing knowledge structures is more effective than making the student create new ones (Davies, 1994, p. 703).

McKay (2000) describes a system for teaching programming which starts with much of the problem solved and, as the student succeeds through the exercises, the amount of problem which is pre-solved is reduced to zero. This is similar to the “fading” technique used in other contexts. This approach is also advocated by Buck & Stucki (2000) [97]. This is the approach taken by one of the instructors who – without referring to scaffolding – scaffolds the students as shown in the following quote:

> *Here are the steps that I follow to help the students achieve this goal: (1) step-by-step, I list how to type, compile, fix and execute a program. The students will have to do this step several times with different programs. I give the students the code to the program. I feel that the students learn the editor and the structure of a program. (2) The next step, the students will fill in the blanks of a program. I will give them partial code to a program and then they will complete the program. (3) The third step, the students will have to write a program, but I give the students the flowchart and the output. (4) Finally, the student will flowchart and write a program on their own. [Later] I try to have them think by writing a lot of small programs that build on one another.*

McKay’s and Buck and Stucki's approaches stand in direct contrast with, for example, Cohen and Boisvert (2004), who advocate and approach which “allows the student to progress from the whole to its parts rather than the reverse" (p. 225).

Senior students

The idea of using senior students to help junior students is well rooted in the concepts of the zone of proximal development and *scaffolding*, as well as the notion of the value of *student interaction with peers*. Heaney and Daly (2004) propose using second-year students to supply feedback to first-year students (see also section on *feedback*). They note four advantages to this approach:

- By taking only the top 10% of the previous class, the senior students have mastered the previous material.
- Students who have recently completed the previous course are well aware of the problems and pitfalls involved, and can help the junior students work through them.
- By taking the top 10% of former students, it is possible to maintain a 1-10 mentor/student ratio, assuming enrollment is relatively constant.
- Senior students who have mastered the material can act as role models for the junior students, which may increase the performance of each generation of student.

They found that using their approach resulted in an improvement in learning for those students who were at risk of failing. One advantage of this method not mentioned by Heaney & Daly is that the senior-level student could benefit from this activity by developing stronger notions of “good” and “bad” programs.

Student learning

Student learning is a goal state in this model. As mentioned earlier, the notion of student learning assumed in this model is primarily cognitive, but also incorporates aspects of andragogy and constructivism as well. An assumption of this model is that, properly done, a combination of student learning, retention, and assessment will result in overall success.

Teaching presence

Good teachers, whether DE or F2F, affect students’ attitudes toward, as well as their knowledge of, the subject they teach. An instructor who is disengaged can only have an effect on a student via the power differential, which will not last past the end of the semester. Sometimes good teachers fear that the transactional distance of DE will destroy their ability to make a real difference in the lives of their students. Anderson refers to this as *teaching presence*, which he claims that it is a critical aspect of successful teaching; he recounts three roles of the instructor: organization, implementing activities to encourage discourse, and direct instruction[98].

If one views education as a shared experience for the entire class, then the role of teacher presence, whether F2F or DE, can not be ignored.

Practical aspects

Please refer to diagram 4-3; this diagram describes the “practical” elements of LP/DE and their interrelationships. By practical is meant those aspects of the domain which have to do with course and time management, methods of assessment, and the primary outcome factors of student success and retention/completion. There are thirteen elements in this part of the model.

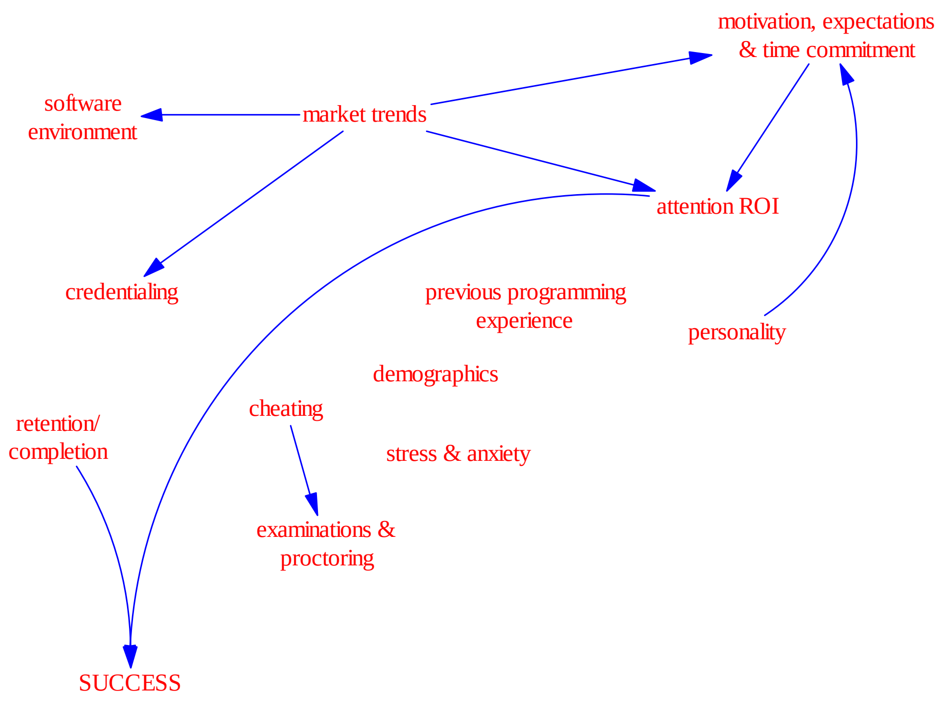

Figure 4-3: Practical elements of LP/DE

Attention ROI

Blackwell (2002) proposes a model of programming using an *attention investment* approach: decisions to program something are made based on (a) the amount of work involved in doing the task w/o programming; (b) the amount of work it will take to program it; (c) the risk that the program will not work; (d) the amount of attention-time saved after the action has been programmed.

For example, consider setting up automated bill payment with one's bank, which is a rudimentary form of end-user programming. There is an initial cognitive (and time) cost to setting up the system – one will need to figure out how to use the web site or phone-response system – but there is an expected pay-off in “attention units” in the long term because one will no longer need to sit down and write checks, etc. each month.

This is similar to the knowledge compilation / automated action response theory of cognition in general. People who cognitively automatize a process benefit from the lower attentional cost – for example, someone who automatizes their drive home from work can listen to the news, a book on tape, or just think about other things. As long as the process works, then there is a pay-off; however, if the automated process fails (e.g., someone drives to their old house after having moved), then some remediation will need to be done, at some cognitive (and time) cost.

In both cases, the decision to program something involves *cost, investment, payoff* and *risk*, and can be seen to model economic decision making, as

Blackwell explains: “Many programming activities promise, through automation, to save attentional effort in the future. The irony of this abstract approach is that the activity of programming may involve more effort than the manual operation being automated. Most decisions to start programming activities are based on an implicit cost-benefit analysis” (Blackwell, 2002, np). Blackwell’s model is an economic account, where the “currency” is mental attention [99].

Blackwell’s model is designed to explain programming behavior in a non-educational environment. However, LP/DE instructors can use this theory to understand much of the behavior of their students. It can be used to help explain why a student would cut-and-paste code over and over for an assignment rather than write a reusable function, and it can be used to explain why a student would continue to pass values by reference when there is no need to do so. Both of these are examples of actions by a student who has calculated that the amount of effort involved in learning the added abstractions is not worth the benefit derived. Armed with this knowledge, the instructor can attempt to change the student’s mind, by attempting to explain why it is important to get over this hurdle or, if all else fails, to entice the student with a threat of failure if the concepts are not mastered.

Cheating

Cheating is a major concern in all LP courses. Woit and Mason (2003) review the literature on cheating in LP courses, noting the following:

- In one study, 34% of computer science students had copied a majority of an assignment from a friend.
- 52% reported that they had collaborated on an assignment that was meant to be individual work.
- 90% of students would not report cheating by others if they knew about it.
- More broadly, 88%-90% of undergraduates admit to cheating at some time in their school careers.

Given this level of cheating in traditional courses, it seems odd that this is one of the first objections that F2F faculty raise with respect to LP/DE. Clearly, cheating is possible (perhaps even likely) in any non-proctored environment, whether it is DE or F2F. Instructors (and perhaps institutions) must decide what the risks and rewards are of using certain types of *assessments*. For example, the computer science and engineering department of The Pennsylvania State University at University Park has recently adopted a policy that 40%-60% of a student's grade must be determined in a proctored setting in the introductory programming course; this policy is currently being incorporated into the first-ever LP/DE course offered by that department.

One type of cheating which is often brought up is that a student could look for an answer on-line, as described by a student:

I tried to cheat by looking up one of the programs on Google. It was a guessing game, so I typed "guessing game Java programming" into

Google. A few similar programs came up, but they used a different source of code so I couldn't use them in my program. I tried examining them to get a better understanding of what mine had to look like, but it only confused me and took me off track.

The instructor must ask him- or herself if the notion of the student researching the problem in order to find a solution is a positive or negative factor. This problem can be ameliorated, however, by assigning idiosyncratic programming problems instead of "the old standards." As one student said:

Overall, I think cheating is hard, because you can't find the identical program you need, but I think finding similar ones can be very helpful in doing your assignments.

Credentialing

Many students see the value of their education in the development of a marketable credential. For example, when asked about the potential for cheating in an LP/DE class, one student responded:

If students complete the coursework, and then successfully take advanced coursework, or land good jobs, cheating is probably minimal and it may not be worthwhile to use limited resources to reduce cheating further. If students do not perform well, the school has a serious problem. The problem might not be that the students have cheated in their coursework. But the problem might also be that the school needs to develop a better curriculum and find more competent instructors.

However, the concept of the economic value of college-level education is not as “common sense-ical” as might be believed. There may be a number of reasons why higher education has economic value. Arai (1998) lists (without necessarily advocating) some of them as:

- Perhaps people with college degrees are more intelligent on average than those without them, and thus the connection between degrees and income is correlatory, not causatory.
- Employers believe a college degree signals intelligence and will therefore place a higher value on an employee with a degree.
- Higher education increases a worker’s productivity (human capital theory).
- Signaling, which is “based on the idea that the role of the college degree is not to certify the knowledge and skills acquired in college, but simply to convey information to society about the degree holder's productivity… According to this theory, those who have high productive capabilities go to college so that firms can identify degree holders (or highly educated people) as more productive and pay them more” (Arai, 1998, p. 47).

Sometimes an employer needs an employee with certain skills and abilities and is not willing or not able to train these employees. In these cases, the employer is looking for some assurance that a potential employee will be able to do a particular job with minimal extra training costs. In these situations, the role of higher education is that of granting *credentials* to the student.

The notion of the credentialing role of the university spotlights the dichotomy between vocationally-oriented and a classic liberal education, and the debate continues to this day [100]. In an LP/DE context, this can present itself in the disagreement of the goals of a CS1 course: some believe it is to teach students to program in a particular environment, others argue that the role of the course is to train students to think. While thinking is undoubtedly something that many employers value, it is not always the case that they are willing to pay a premium for an employee who is credentialed to do so. Thus the debate rages on.

Demographics

One of the most commonly studied aspects of LP/DE is demographics; the assumption behind this is that certain differences between students will be a predictor of their potential for success in the course. A significant proportion of the studies performed on LP (especially in the late ‘80s and early ‘90s) were correlatory studies of demographics and LP. However, this notion is problematic for two reasons: (1) the data is not clear, and (2) even if it was clear, there are no actionable ramifications for practice. One of the few wide-ranging demographic studies of LP/DE was McGill, Volet and Hobbs (1997), who found:

- no significant relationship between age and course completion.
- no significant relationship between gender and course completion.
- no significant relationship between first-time DE students and

course completion.

- no significant relationship between a student being a computer science major and course completion.
- a significant relationship between entrance exam scores and course completion.

The last point indicates that there is a correlation between students who are doing well and those that finish the course, which seems intuitive. Whereas it may be possible that non-academic factors will affect a student's decision to withdraw from a class, it also seems relatively certain that they are not likely to withdraw if they are doing well.

The result reported in the section on *pre-assessment* bears repeating here; that is, that even if demographics and background is known to effect student achievement, this has little value to a particular student and/or the instructor, since demographics and background are "given" and not under the control of the student or instructor. Perhaps from a policy standpoint an institution can decide to only admit students who have a high likelihood of succeeding, but this is outside of the power of the student and instructor to effect.

Examinations and proctoring

The inability of students to write programs after finishing an LP course is a perennial topic of conversation among instructors. This problem was underscored by a student who wrote:

I read the book and understand what it is about, I do well on the tests but I seem to have a hard time applying it… I spend most of time trying to apply to a program what I am learning. I think I have it and then the program doesn't work and I have no idea why. I cant seem to pick out my errors.

Califf and Goodwin (2002) found that students were passing their CS1 course without the ability to program, and thus they decided to begin testing students at the end of a course to see if they could write, compile, run and debug a program [102]. Although this sounds straight-forward, there is disagreement over the proper goal and content of LP/DE courses and, this being the case, it is also inevitable that there would be disagreement over the appropriate types of assessments, including the ecological validity of examinations. Traditionally there have been two types of LP timed, proctored tests: (1) multiple choice and (2) hand-written, short answer.

Multiple choice exams are preferred by many instructors due to their objectivity and ease of scoring. A multiple choice exam might focus on the declarative knowledge portion of the course, requiring the student to identify valid data types, specify the parts of a computer, etc. More task-based multiple choice exams might give the student a program segment and require that the correct output be identified. Even more complex exams could present an almost complete program and require the student to select the appropriate option to successfully complete the program. However, multiple choice exams can be seen as ecologically invalid if the goal of the course is to teach the student how to program.

Short-answer exams, where the student is required to produce a short segment of program code, are more task oriented than many multiple choice alternatives. In these kinds of exams, a student is asked to write a function or procedure which, given certain input, will produce a particular output. These types of exams have the advantages of: (1) being able to be completed within a standard class period, (2) requiring a reasonable amount of time to grade (while significantly more than a multiple choice exam), and (3) demonstrating that the student can produce program code. However, these kinds of exams still suffer from validity problems: (1) students have been typing (not hand-writing) programs all semester, with the aid of the program compiler (which indicates syntax errors in a program), and thus the task performed is not the task being tested; (2) the student is not required to demonstrate that s/he can create an entire working program, only small bits of them.

A third type of exam is the proctored lab (what McCraken, Almstrum, Diaz, Guzdial, Hagan, Kolikant, Laxer, Thomas, Utting and Wilusz (2001) refer to as a *charette*). In these exams, students are required to write one or more programs in real time, usually in the same environment in which they did their assignments. This approach has the advantage of testing the ability of the student to write a program from scratch, but suffers from the following problems: (1) students are subject to time constraints for practical reasons, which may indicate ecological invalidity, (2) such examinations can be very stressful for the student, and (3) the results are hard to grade. In addition, access to the Internet and other electronic resources may encourage cheating on the part of the student.

Some oddities exist in assessment in LP/DE. For example, Zachary and Jenson (2003) designed their introductory programming course so that each of the twelve lessons includes a on-line example-based narrative, a series of programming exercises, and an assignment. However, in a somewhat surprising move, they base course grades to a large extent on proctored, *paper-based* exams.

Given these problems with assessment, Shaffer (2005) developed and tested a web-based system which allowed students to check programs for syntax errors and correct output before submitting the programs for grading. This allowed the students to focus on the problem at hand and not on extraneous details (minimizing cognitive load). Also, since the students used the same interface for assignments, quizzes and major exams, the amount of stress test-taking was lowered. An additional benefit of this approach is that the exams were graded automatically, with only a visual review by the instructor necessary.

DE programs often find that requiring proctored exams is administratively difficult. For those students who are quite distant from the campus, proctors are often drawn from local teachers, clergy or employees of testing centers.

<u>Market trends</u>

According to Cohen and Boisvert (2004), “In the past [computer science departments] had little interaction with industry partners that might employ their graduates. This has led to each institution deciding what is most appropriate for themselves without regard to the needs of the students who want to move from community college to university or to the employer needing to enhance the workforce" (p. 225). However, market trends are an important source for course content and thus the choice of programming language used [103]; for example, one student in the study said:

I am taking this class because I wish to re-enter my former career in some capacity. There have been lots of changes in the profession and this is my first step to figuring out how much has changed and what direction I want to take with my career.

Another wrote:

I want to learn more about programming because I believe it will make me more marketable when I look for a job in my major.

And another simply said:

The reason for me taking this course is that my job requires it.

Another aspect of market trends is competition among educational service providers. One student in the study produced a quite impressive analysis of his decision to attend his local community college which, though long, deserves inclusion here:

Right now, this course costs me $240 in tuition and fees and about $150 *for books for a total of $390 for the class. However the real cost is closer to $1050. It does not cost me that much because the state subsidizes the community college for $300, the county government subsidizes the community college for $300, and the federal government will give me 20% of my direct tuition costs of $300 that equals $60 through the lifetime learning credit. Realize I would never, ever pay $1050 for this course. There are for profit corporate giants like Microsoft that have distance learning courses that do not cost so much and there are reputable Universities – I'm not talking about the University of Phoenix – that offer distance-learning courses more competitively. I have seen some in the $600 range plus books for a total of $750. Suppose, the government decides distance education is too expensive and decides to not only cut subsidies to community colleges for distance learning students but also chooses to subsidize those students directly for their distance learning education costs at any institution they choose just like the federal government does for the lifetime learning credit. If the state and county reduce their total subsidy from $600 to $400, I would happily make another choice. If I take a course at the community college now with this reduced subsidy, it will cost me more: $1050 - $400 from state and county - $130 which is twenty percent of my cost from the federal government with the lifetime credit for a total of $520. However, that other course which is more expensive to me presently at $750 would be cheaper now: $750 - $400 from the state and county - $70 from the federal government with the lifetime credit for a total of $280. It would be vouchers for distance learning for the purpose of increasing competition and reducing costs. I think this will happen because it is unreasonable to charge the same tuition for students who show up in a brick and mortar building, which adds to the cost of education, to distance learners who take far fewer resources to educate.*

It is extremely clear from the data analyzed that the future of LP/DE will be strongly influenced by market factors from both the supply and the demand side. However, some point out that higher education is not supposed to simply be a service industry for potential employers [104].

Motivations, expectations and time commitment

As compared to demographics, the motivation and expectations of students has been found to be a more consistent predictor of retention and course success. The student for whom the LP/DE course has relevance (for example, is central to his or her career goals or major) and who both is motivated to and expects to do well is more likely to succeed.

Rountree, Rountree, Robins and Hannah (2004) found that students who thought at the beginning of the course that they would not get an "A" were at risk for doing poorly (C or lower), and suggest "that course advisors spend some time explaining to these students the level of commitment required of our students and how necessary it is to be aiming for mastery of the skills involved rather than merely aiming to pass" (p. 103). For example, one instructor states:

> *"I send students an email prior to the course beginning and tell them what is expected of them within the first 2 weeks. This gives them a heads up on buying books and begin reading and installing software. I have a discussion board set up for them to introduce themselves to others, ask questions, and chat about their expectations and fears. I have a chat during the second week so that they can get to know each*

other and me."

However, students do not always have a strong idea of the required time commitment for a course, especially a DE course. This is borne out by the following quotation from a student in the study:

> *I am taking this particular course online because of time constraints only. I believed that it would be more convenient and save me a lot of time. This was a poor assumption. I actually spend much more time than I would if I attend in the traditional format.*

Another student states the following, apparently unaware of the "industry standard" approximation of three hours outside of class for every credit hour – which would be twelve hours per week for a three-credit DE course (including the three hours per week "in class" time):

> *The worst thing about [the course] is the enormous amount of time it takes to complete the assignments and the awful impact it is having on my ability to spend time fulfilling familial responsibilities. I spend about 2 hours/day on this course, seven days a week. I believe that this is an unusual amount of time to invest in any course, and I attribute this to the lack of teaching/instruction that I am receiving by the professor.*

Pallof and Pratt (2003) further suggest that instructors discuss time commitments with students during the first week of the course. But not all time spent is well spent, and neither is time spent necessarily directly correlated with success. For example, Katz, Aronis, Allbritton, Wilson and Soffa (2003) found that time spent on a programming tutorial was negatively correlated with increases in score from pre- to post-test; “suggesting that students who took longer to work through the tutorial as whole and individual chapters were having difficultly grasping the material” (p. 160).

Levels of motivation and commitment can change during a semester. For example, one student noted:

> *I took the course because it was relevant to my major when I signed up for classes. I have since changed direction in life, and the only reason I’m still taking this class is so I can still be a full time student.*

Many students, especially non-traditional students, have time commitments outside of school; these might be family or work responsibilities, as the following responses from students in the study indicate:

> *I like distance learning because it is at my own time and pace and also it is very helpful to me in balancing my kids, home and work.*

And:

> *My work schedule changes every week.*

And:

> *I am a single divorced mom of a 12 year old that I live with and I also recently moved home to help take care of my parents who need some help also. I work fulltime and take as many classes as I can. This semester I am only taking one course due to the difficulty of it. I also keep busy with many other things and find that computers are a wonderful way to make time for everything in my life.*

… or they might be related to a specific situation:

I am taking this as a distance learning class so I can drive my boyfriend to and from work until he gets his license back. The only time I could have fit the traditional class into my schedule would interfere with this.

Previous programming experience

Previous programming experience can be both a blessing and a curse with regard to LP/DE (see also the section on *transfer and interference*). For example, a number of studies (e.g., McGill, Volet and Hobbs, 1997 and Hagan and Markham, 2000) have shown that previous programming experience is a positive indicator of success for a student of LP/DE. Holden and Weeden (2004) found that although prior programming experience was a factor in success, there were decreasing marginal returns on more prior experience [106]. They further found that after the initial course, there were no significant effects on grades based on prior experience. Morrison and Newman (2001, p. 180) found “extraordinarily high confidence” (higher than 99.9%) of such an effect, but also found that courses taken at a four-year educational institution were better at preparing students for a CS1 course than junior college or high-school level experience.

However, previous programming experience can result in *proactive interference* and may cause difficulty for the student in learning the new material.

The “previous experience” might not even precisely be programming experience, but simply experience with other language formalisms. Buck & Stucki (2000) point out the “cognitive mismatch” students have when learning about assignment statements; there is a tendency to extend the notion of the equals sign from algebra into programming and thus to misunderstand its use in programming.

Retention / completion

Completion is the property of a student completing a course, while *retention* is the property of a student staying within a program. Retention is an important metric for administration of LP/DE, since the future of a course of study and/or an entire institution may hinge on this variable. Course completion is an important component of a student's experience, since failure to complete a course may result in a major life change (changing majors or perhaps dropping out entirely). The two notions of retention and completion have been combined into one variable here because a student who does not complete a course is less likely to continue within a program of study. (NB: This is an example of trading precision for power of the model.)

Kleinman & Entin (2002) find that there is a higher drop-out rate for DE versus F2F courses based, according to the authors, on a “technology hurdle” wherein students had difficulty getting their software and hardware set up correctly. They found that for an introductory programming class that online students tended to drop out quicker, primarily due to intial problems setting up their equipment, whereas the drop-out rate in a F2F class was more gradual and lower overall.

One example of a student illustrates the snowball effect that can occur for an LP/DE student:

> *The worst thing about the class is that you are not right in front of the prof. getting answers right after you ask them… I really don't even know the instructor, other than the prof is a she. The only communication that we had is through e-mails… I don't even talk to the other students. I don't even know their names or why they are taking the class… There is a bulletin board that we can use on our course compass but I never saw anything posted… I ran into the problem of not understanding how to do a program. I did the program and handed it in but it was still wrong. I tried looking things up in the book but it didn't help me at all… The class was a lot harder than I expected for being from a local community college. I took C++ programming before and this class just seems a lot harder because there is no direct contact between you, your instructor and your other students. The hardest part of the class is writing the programs, I don't think there is exactly any easy part… Definitely spending more time than I thought. I knew the class wasn't going to be very easy but I did expect it to be a little more easier… Most of my time for this class is spent writing the program instead of reading. If you don't start them early then you are not able to ask her questions.*

Right after this message was sent, this student dropped the class.

There is a common notion that some students "get" programming and the others never will (see the section on student development stage). However, this notion may be arbitrarily limiting the number of students who stay in a programming course. One student put this quite well:

> *I have taken programming before and understand the feeling of not getting it. While you have the feeling, it is just terrible. But I know it does not matter, and eventually I will get it. Often while not getting it, you learn a lot about error messages and logic errors. What is important is that you eventually get it and you do not give up. It's like learning to walk. Some babies start at 12 months, some earlier, and some even much later. However once you learn to walk, it does not matter when you started. You do not pick a track star according to when they started walking. You don't pick a good programmer according to when they "got" programming.*

Software environment

There are two major categories of software environment germane to this discussion of LP/DE: course management software and development environment. Course management software (CMS) has become fairly stable and mature over the past few years, and not much will be said about the standard aspects (e.g., message board, grade posting, etc.) here. However, there are some aspects of a CMS which are specific to LP/DE; for example, when asked to describe an ideal software environment for LP/DE, one instructor said:

> *I would then like to correct their Java program while verbally and visually explaining what I was doing. Click a single send button that would send the audio and video to the student and a copy to my files.*

In other words, I would like to demonstrate how to fix their particular programs so they could visually see it as they would in a face-to-face lab.

The second software aspect of LP/DE is the choice of development environment (the software used to develop the programs, often called an IDE or interactive development environment) is a crucial aspect of any LP course, and can “make or break” the success of an LP/DE course. Ko and Myers (2003) point out that poor choice of programming interfaces can cause attentional strain and force premature strategic decisions (p. 14). Several sources note that the details of operating the programming environment, though necessary to complete the course tasks, are usually not the subject of the classes themselves and often are never discussed [108] [109][110].

Emory & Tamassia (2002) describe their Java Environment for Remote Programming Assignment (JERPA). Their program attempts to shield introductory students from much of the inner workings of software development tools in order to “teach the principles of object-oriented programming without overwhelming the students with the technical idiosyncrasies” (p. 307).

Students sometimes press to use “real” IDEs in their classes so that they can use this experience directly at their jobs or put the experience on their resumes. However, just as professional *programming languages* may not always be appropriate for student use, professional IDEs also may not be appropriate for student use, given that the number and variety of options afforded professionals may overwhelm a student. One instructor in the study said that an ideal software environment would include:

> *Something that I wouldn't have to deal with how each student's computer is set up - having to install the compiler is difficult to deal with.*

This sentiment is echoed by a student in the same study, who elaborates as follows:

> *I cannot view the videos for the lectures. I do not have a soundcard on one of my computers and I do not want to bug our computer tech for permission to download software to view them. Had I known before class started, I would have worked hard to put things in place to view the lectures. I have not resolved it… In the beginning of class it was horrible how many students were having problems. Everything from hardware, software, textbooks and navigating around in the [CMS] environment. I thought [the college's information systems] department should be utterly ashamed of themselves. Many of the problems could have been prevented if only the [information systems] department clearly stated what was necessary for the class. I suggested on the message board they hold a free optional seminar explaining what was necessary, showing the equipment, and what software would need to be installed before classes even started. I suspect many got off to a bad start and are still suffering because of the initial problems.*

Given the high risk of early drop-out in LP/DE courses, use of complex IDE software may be contra-indicated. Heo (2003) and Shaffer (2005) both describe an LP/DE interface which runs entirely through a web browser and thus is available to any student without the need to install and support extra software.

Stress

Student stress comes from many dimensions of the student’s experience, many of which are outside the control of the course instructor. For example, one student was showing clear signs of stress late in the semester:

> *My husband is doing training for his work and needs the computer the same time as I do most of the time, so it has been a bit frustrating because we only have one computer and both need it. I work throughout the week so sometimes it is difficult to get my assignments done. It is even more difficult to get to bed at a normal hour also because at times, we both need the computer, and one of us always suffers. I don't know how many nights I just said: "I've had it, I am going to bed because I am not getting anywhere." Then of course I would take my book to bed and read it anyway until I got frustrated even more and fell asleep. It is so hard for me to concentrate when I am tired. My mind goes blank when I am tired... I have found myself trying to get my homework done at work on my spare time when the due dates for my assignments are due in the middle of my work hours because like I said, sometimes the night before, I still didn't get anywhere...I'm not even saying that I waited until the night before to do it either.. It's just that at times, I don't get anywhere even days before...It is hard for me to come up with the logic of the programming assignment because of working at the same time. I am the only one in my office at work. My boss works out of his home and visits rarely so sometimes I try to do some here and there. My boss gives me things to do and if I get done early, I might try and do some homework. It sucks, because homework is so much easier to do during the day and I*

don't get much sleep at night because I have a problem falling asleep until late...I work in the day, so it puts a damper on things.

However, it is common to assume that, everything else being the same, minimizing student stress is important. Certain aspects of course design can add stress, but of course the benefit may outweigh the cost; that is, the ultimate non-stressful situation is not to be graded at all, as with a lifelong learning class or public seminar.

On-line tests of programming ability can be quite stressful; for example, Woit and Mason (2003) note that 15% of one group of students studied complained to the instructor that they did not work to their potential because of the stress of the online test; however, a different group of students, who had had weekly on-line quizzes, had no such complaints.

Avoiding high-stakes testing by giving the student many opportunities for grades during a semester can lower the stress caused by any one particular assessment. Woit and Mason (2003) suggest discarding 35-50% of the lowest grades; this may be problematic, though, if the student does well at the beginning of the course but never masters the later material.

Success

The ultimate goal of this model is student success; note that whereas many variables feed into this one, none are fed by it. According to Katz, Aronis, Allbritton, Wilson and Soffa (2003), "A complex array of experiential, affective, personality, socio-cultural, and cognitive factors have been shown to predict achievement [in computer programming] – for example, simply owning a computer; using a computer in pre-college computing classes; prior programming experience; confidence, intrinsic motivation, and having clear career goals; and various aptitudes, such as math ability, spatial reasoning ability, verbal reasoning ability, and Piagetian formal operations" (p. 157). This model attempts to capture and document as many of these variables as is practical.

Entire model

Diagram 4-4 shows the model in its entirety. With fifty-one variables, this model can be daunting; ideally the interested person will study the diagram in its own right, and not just as a way of breaking up the monotony of the text. As stated at the outset, this model is meant to be viewed as one would a map of a city; the reader might be looking for particular features, or may just be trying to get "a lay of the land." Readers of the electronic version of this document are encouraged to use the PDF zoom tool for a more detailed view of the model; also, hovering the cursor over a model element will pop up a brief description of the variable.

The diagram enables the reader to see that "all roads lead to success"; clearly, important goals of an LP/DE program include institutional and societal success; however, this model's focus is on student success. It is hoped that the organization's and society's goals are aligned with the student's and thus focusing on the success of the student will entail success for the institution and society.

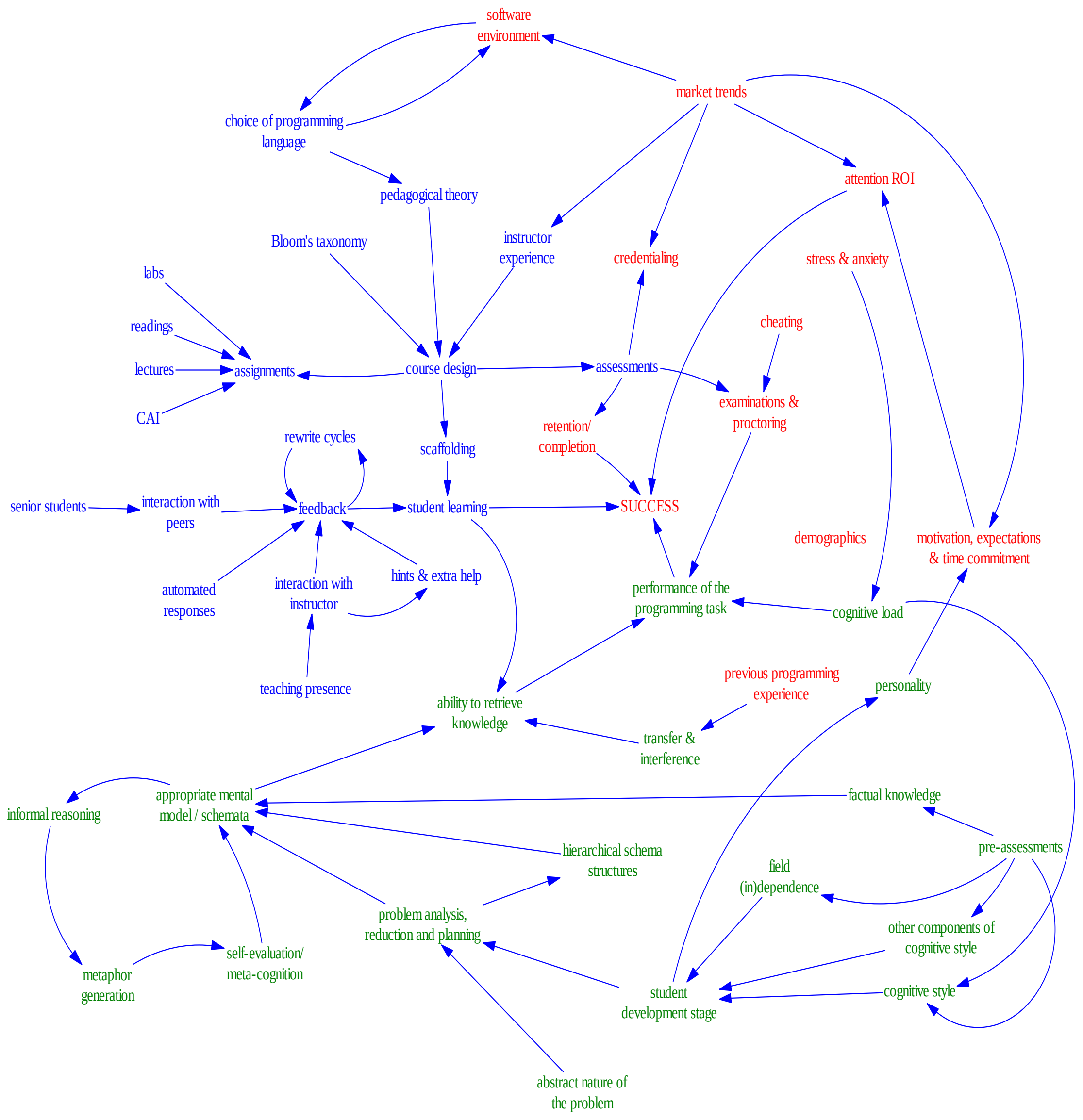

Figure 4-4: Entire model of LP/DE

The goal of the study was to create a deep model of LP/DE and, in order to obtain the requisite depth, only the cognitive, pedagogic and practical aspects were considered. Socio-economic and organizational factors, while very important, are left for future work. In addition to informing the practitioner, the model presents a structure for future research work; by selecting any two variables, one can ask: what is the relationship between these two variables? Is the relationship direct or indirect (requiring another intervening variable between the two under consideration)?

Chapter 5

Conclusions and discussion of future work

Learning to program via distance education (LP/DE) is a complicated affair, involving cognitive, pedagogic and practical variables in a complex interaction which defies simplistic modeling. One of the most basic results of this project has been that LP/DE is not for everyone; as one student in the qualitative study succinctly put it:

This is a terrible way to learn programming. These concepts require instruction and guidance. This online course offered no instruction at all.

This might indicate that the LP/DE course was not well-designed; however, it also might indicate that there is an aspect of LP that is very difficult to relay via DE. LP is hard in any format, as Thomas et al indicate as much when they conclude that:

"Many students seem to be put off by introductory programming courses. They either fail and give up on the subject or they continue with their degrees but vow that their future careers will not include programming… We may not always do so, but there is no reason to suppose that we cannot remedy some of these problems in introductory programming modules and make our courses more interesting and appealing to the wider range of students who now enter higher education… The purpose of a university education … is to extend opportunities, rather than to narrow them" (Thomas, Ratcliffe, Woodbury, Jarman, 2002, p. 33).

One of the most important aspects of LP/DE is the bimodal distribution of grades – what many instructors call the student's ability to "get it." The best explanation for this seems to be a combination of the student's developmental level and cognitive style. It seems that those who are best at "getting it" will thrive in a LP/DE environment, while those who are less oriented toward programming will struggle with the class. This is not a deep nor a surprising result, but it could well be a useful one. Some form of pre-test which measures preparedness for an LP/DE course could be developed (as several studies have suggested); it is an open question as to whether this should be prescriptive or merely suggestive. Such a test – instead of focusing on non-directly-measurable constructs such as "developmental level" – could instead be task-oriented, requiring the student to perform a certain series of tasks in the same environment that s/he would be working in for the actual course. Once several semesters of data correlating this pre-test and final grades is obtained, it should be possible to predict with some level of accuracy how well a student will do in the LP/DE environment. Based on how the student performs on this assessment, suggestions could be made as to whether LP/DE is appropriate. However, DE is not a choice but a necessity for some and therefore simply throwing up one's hands for those students who are not ideally suited for the task is not appropriate educational policy.

Another clear result from this study is that instruction in LP/DE needs to take into account each student's individual differences and thus can not be designed in a one-size-fits-all manner. Even those students who are able to "get it" do not all learn in a monolithic manner, as Foreman points out: "The results of the current study suggest that individual differences should be considered in programming instruction regardless of programming language and age. Adapting instruction to every individual is not feasible. However, alternative instructional approaches would help different styles of learners, of different cognitive styles and aptitudes, to learn programming skills more quickly and effectively. The interaction between individual differences and programming instruction needs to be understood in order to improve programming instruction" (Foreman, 1990, p. 60). Optimum LP/DE course design will include multiple ways for the student to learn the material; perhaps in a tutorial mode for some, in a didactic way for others, and in a discovery way for still others. As one student in the qualitative study put it:

> *I have sometimes wondered how people learn this subject as I have to reread the texts many times and do extra smaller problems to understand how to do the big thing.*

Perhaps students would prefer to mix and match their own combinations in order to best engage themselves with the material. This kind of course development takes a large investment in development time, but, if done well, could result in a powerful course scheme which could be re-used for courses based on a variety of programming languages.

It is still unclear what the appropriate approach to teaching an introductory programming class is; should the instruction focus on an explanation of how the computer works and interprets programming commands, or should the programming language be taught as an abstract notational machine (as discussed in Arzarello, Chiappini, Lemut, Marara, and Pellery, 1993)? In the past I have used the latter approach, but as a result of this research I am no longer convinced of its efficacy.

Teaching programming as a kind of cognitive apprenticeship highlights the task orientation of the subject. For example, an instructor could solve a programming problem in real-time while other students looked on, either via a captured video or by using video conferencing products such as “go to meeting” (www.gotomeeting.com) where multiple people can view one person’s PC screen from remote locations. This approach differs considerably from the approach wherein already designed, coded and debugged programs are made available to the student; this latter approach has the effect of masking the *doing* aspect of programming in favor of a sort of *received wisdom* model. The instructor “must single out ways and forms of novice-expert interplay, able to favour the development of cognitive and metacognitive processes which characterize LP as a cognitive apprenticeship” (Arzarello, Chiappini, Lemut, Marara, and Pellery, 1993, p. 293).

The effects of cognitive load are clearly important to LP/DE, and it is important to attempt to minimize extraneous cognitive load in the development of LP/DE courses. For example, the instructor can assign the same problem for several weeks in a row, requiring that it be done in different ways, so that the student does not have to expend mental energy in understanding new problems each week.

The value and future of research in LP/DE

Heo suggests that “research has a clear mission to provide the distributed education instructor and student an effective transition from the traditional classroom to the web-based educational environment” (Heo, 2003, p. 152). This is not to dismiss the value of, for example, controlled experiments in cognitive psychology; indeed we need the results of such studies to inform a pedagogical model such as that provided in this paper. However, from the standpoint of supporting educators – in this case, LP/DE instructors – we need to keep the bigger picture in mind. System dynamics modeling enables this. Creating models like that presented in this paper can serve multiple purposes: (1) it can act as a “road map” for those traveling through the world of LP/DE, and (2) it can be used as the starting point for developing computational simulations of educational systems. As Altman (2001) points out, “computational simulation is perhaps the only reliable vehicle for bringing cognitive theory to bear on complex behavior” (p. 190). With computational simulation it is possible to test dynamic theories of the interactions between LP/DE variables. Such an approach enforces precision in the models, since the variables do not “become obvious until one tries to simulate performance computationally, at which point it becomes clear that the model cannot perform without them” (Altman, 2001, p. 195).

Given the 51 variables presented in this paper, there are over 1200 potential interactions between these variables, and each of these could be the subject of a detailed study in-and-of-itself. Whether this is reasonable is of course an economic decision; specifically, if we do this detailed analysis, what will it buy us? Is optimizing LP/DE a socially worthwhile goal? Assume for the moment that this research managed to highlight those 20% of variables of LP/DE which cover 80% of the effect; is 80% good enough? Given that the curve is exponential, perfection will be infinitely costly, so where is the cut-off point for the return on investment? These questions are outside of the scope of this paper, but must be answered at some level before more resources are put into studying the phenomenon which is LP/DE.

REFERENCES

Ally, M. (2004). Foundations of educational theory for online learning. In T. Anderson (Ed.), *Theory and practice of online learning* (pp 3-31). Canada: Athabasca University.

Altman, E. (2001). Near-term memory in programming: a simulation-based analysis. *International Journal of Human-Computer Studies*, 54, 189-210.

American Association of Community Colleges (2005). Fast Facts. Retrieved September 14, 2005 at http://www.aacc.nche.edu/Content/NavigationMenu/AboutCommunityColleges/Fast_Facts1/Fast_Facts.htm

Anderson, T. (2004a). Toward a theory of online learning. In T. Anderson (Ed.). *Theory and Practice of Online Learning*, (pp 33-60). Athabasca, CA: Athabasca University.

Anderson, T. (2004b). Teaching in an online learning context. In T. Anderson (Ed.), *Theory and practice of online learning* (pp 273-294). Canada: Athabasca University.

Anderson, J. and Skwarecki, E. (1986). The automated tutoring of introductory computer programming. *Communications of the ACM,* 29 (9), 842-849.

Anthony, J. and Payne, M. (2003). Group dynamics and collaborative group performance. SIGCSE'03: Conference of the ACM special interest group on computer science education. Reno, Nevada, USA.

Arai, K. (1995) *The economics of education: An analysis of college-going behavior*. Tokyo, Japan : Springer.

Argyris, C. and Schon, D. (1974). *Theory in practice: increasing professional effectiveness*. San Francisco, CA, USA : Jossey-Bass.

Arzarello, F., Chiappini, G., Lemut, E., Marara, N. and Pellery, M., 1993. Learning programming as a cognitive apprenticeship through conflicts (1993). In E. Lemut, E., B. Du Boulay, B. & G. Dettori, G. (Eds.), *Cognitive models and intelligent environments for learning models* (pp. 284-297). Berlin, Germany: Springer-Verlag.

Banathy, B. H., & Jenlink, P. M. (2004). Systems inquiry and its application in education. In D. H. Jonassen (Ed.), *Handbook of research on educational communications and technology, second edition* (pp. 37- 57). Mahwah, NJ: Lawrence Erlbaum Associates.

Bishop-Clark, C. (1995). Cognitive style and its effect on the stages of programming. *Journal of Research on computing in education*, 27 (4), 373-387.

Blackwell, A. (2002). First Steps in programming: A rationale for attention investment models. *IEEE 2002 Symposia on Human Centric Computing Languages and environments*.

Bruce, K. (2004). Controversy on how to teach CS1: A discussion on the SIGCSE-members mailing list. *Inroads – The SIGCSE Bulletin*, 36 (4), 29-34.

Buck, D. and Stucki, D. (2000). Design early considered harmful: Graduated exposure to complexity and structure based on levels of cognitive development. *ACM Special Interest Group on Computer Science Education*

2000. Austin, Texas, USA. 75-79.

Califf, M. and Goodwin, M. (2002). Testing skills and knowledge: Introducing a laboratory exam in CS1. 2002 Conference of the ACM Special Interest Group on Computer Science Education, February 27-March 3, Covington, Kentucky, USA.

Capretz, L. (2003). Personality types in software engineering. *International Journal of Human-Computer Studies*, 58, 207-214.

Carbone, A. and Sheard, J. (2002). A studio-based teaching and learning model in IT: What do first year students think? The 7th Annual Conference on Innovation and Technology in Computer Science Education, Aarhus, Denmark.

Cheang, B., Kurnia, A., Lim, A., Oon, W. (2003). On automated grading of programming assignments in an academic institution. *Computers and Education*, 41, 121-131.

Clancy, M., Titterton, N., Ryan, C., Slotta, J., Linn, M. (2003). SIGCSE'03: Conference of the ACM special interest group on computer science education, Reno, Nevada, USA.

Cohen, R. and Boisvert, D. (2004). Aligning programming education between community colleges and universities. SIGITE'04, October 28-30, Salt Lake City, Utah, USA. 224-226

Crutchfield, J. and Mitchell, M. (1994). The evolution of emergent computation. *Santa Fe Institute Technical Reports*. Retrieved September 15, 2005 at http://www.santafe.edu/projects/evca/Papers/EvEmComp.pdf.

Culwin, F., MacLeod, A., and Lancaster, T. (2001). Source code plagiarism in UK HE computing schools, issues, attitudes and tools. Internal report, South Bank University, London. Retrieved January, 2006 at http://www.ics.ltsn.ac.uk/pub/conf2001/papers/Culwin.htm.

Davies, S. (1994) Knowledge restructuring and the acquisition of programming expertise. *International Journal of Human-Computer Studies*. 40, 703-725.

Davies, S. (1991). The role of notation and knowledge representation in the determination of programming strategy: A framework for integrating models of programming behavior. *Cognitive science* 15, 547-572.

Dellas, H and Sakellaris, P. (2003) On the cyclicality of schooling: Theory and evidence. *Oxford Economic Papers*, 55, 148-172.

Dutton, J., Dutton, M. and Perry, J. (2002). How do online students perform as well as lecture students? *Journal of asynchronous learning networks*, 6 (1), no page numbers.

Dutton, J., Dutton, M. and Perry, J. (2001). Do online students perform as well as lecture students? *Journal of engineering education*, 90 (1), 131-136.

Emory, D. and Tamassia, R. (2002) JERPA: A distance-learning environment for introductory Java programming courses. *Proceedings of the Special Interest Group on Computer Science Education (SIGCSE)*, February 27-March 3, 2002, Convington, Kentucky, USA.

Engestrom, Y. and Miettinen, R. (1999). Introduction. In Y. Engestrom (Ed.), *Perspectives on activity theory* (pp. 1-18). Cambridge, UK: Cambridge University Press.

Fay, A. and Mayer, R. (1988). Learning LOGO: A cognitive analysis. In R. E. Meyer (Ed.), *Teaching and Learning Computer Programming: Multiple Research Perspectives* (pp 55-74). Hillsdale, NJ: Lawrence Erlbaum Associates.

Fernandez, E. and Williamson, D. (2003). Using project-based learning to teach object oriented applications development. Proceedings of the 4th conference on information technology education (CITC4 2003), Lafayette, Indiana, USA. October 16-18, 2003.

Fetterman, D. (1998). *Ethnography: Step by Step.* Thousand Oaks, CA, USA : Sage Publications.

Foreman, K. (1990). Cognitive characteristics and initial acquisition of computer programming competence. *School of Education Review*, 2, 55-61.

Forrester, J. W. 1968. *Principles of Systems*. Cambridge MA: Productivity Press.

Frick, T. (1995). Understanding systemic change in education. Retrieved August, 2004 at http://education.indiana.edu/ist/courses/r695fric.html.

Gamma, E., Helm, R., Johnson, R. and Vlissides, J. (1995). *Design Patterns: Elements of Reusable Object-Oriented Software*. New York, NY, USA : Addison-Wesley Professional.

Geertz, C. *The interpretation of cultures.* New York, NY, USA : Basic Books.

Gentile, J. and Lalley, J. (2003). *Standards and Mastery Learning*. Thousand

Oaks, CA, USA : Corwin Press.

Genzuk, M. (1999). Tapping into community funds of knowledge. In: Effective Strategies for English Language Acquisition: A Curriculum Guide for the Development of Teachers, Grades Kindergarten through Eight. Los Angeles Annenberg Metropolitian Project/ARCO Foundation. Los Angeles.

Hagan, D. and Markham, S. (2000). Does it help to have some programming experience before beginning a computing degree program? The 5th Annual Conference on Innovation and Technology in Computer Science Education, Helsinki, Finland. 25-28

Hammersley, M. and Atkinson, P. (2003). Ethnography: Principles in Practice. New York, NY, USA : Routeledge.

Harris, R., Iavecchia, H., Ross, L. and Shaffer, S. (1987). Microcomputer Human Operator Simulator (HOS-IV). Human Factors Society, Annual Meeting, 31st, New York, NY, USA. 19-23 Oct. 1987. pp. 1179-1183.

Hasker, R. (2002). HiC: a C++ compiler for CS1. *Journal of Computing Sciences in Colleges,* 18 (1), 56-64.

Hatchman, M. (2004, August 24). Outsourcing debate rages on. *Eweek*.

Haugeland, J. (2000). What is mind design? In J. Haugland (Ed.), *Mind Design II: Philosophy, Psychology, Artificial Intelligence*. Cambridge, MA : MIT Press.

Heaney, D. and Daly, C. (2004). Mass production of individual feedback. The 9th Annual Conference on Innovation and Technology in Computer Science

Education, Leeds, UK.

Hellriegel, D., Slocum, J., and Woodman, R. (1995). *Organizational Behavior.* Minneapolis/St. Paul, MN : West Publishing Company.

Heo, M. A learning and assessment tool for web-based distributed education. Proceedings of the 4th conference on information technology education (CITC4 2003), Lafayette, Indiana, USA. October 16-18, 2003. 151-154

Holden, E. and Weeden, E. (2004). The experience factor in early programming education. ACM Special Interest Group on Innovative Uses of Technology in Education, October 28-30, 2004, Salt Lake City, Utah, USA. 211-218.

Hoover, H. and Rudnicki, P. (1996). Teaching freshman logic with Mizar-MSE. *DIMACS Symposium on Teaching Logic and Reasoning in an Illogical World*, Rutgers University, Piscataway, New Jersey.

Hudak, M. and Anderson, D. (1990). Formal operations and learning style predict success in statistics and computer science courses. *Teaching of Psychology*, 17 (4), 231-234.

Hulkko, H. and Abrahamsson, P. (2005). A multiple case study on the impact of pair programming on product quality. International conference on software engineering '05, May 15-21, 2005. St. Louis, Missouri, USA. 495-504

Jackson, M. (2003). Why software writing is difficult and will remain so. *Information processing letters*, 88, 13-25.

Jakovljevic, M. (2003). Concept mapping and appropriate instructional strategies in promoting programming skills of holistic learners. *Proceedings of the*

SAICSIT, Gauteng, South Africa, pps 308-315.

Jehng, J. (1997). The psycho-social processes and cognitive effects of peer-based collaborative interactions with computers. *Journal of Educational Computing Research*, 17 (1), 19-46.

Johnson, R. and Onwuegbuzie, A. (2004). Mixed method research: A paradigm whose time has come. *Educational Researcher*, 33, 7, 14-26.

Johnstone, D. and Tate, M. (2004). Bringing human information behaviour into information systems research: An application of systems modeling. *Information Research*, 9 (4), no page numbers.

Kane, T. and Rouse, C. (1999). The Community College: Educating students at the margin between college and work. *The Journal of Economic Perspectives*, 13 (1), 63-84.

Katz, S., Aronis, J., Allbritton, D., Wilson, C. and Soffa, M. (2003). A study to identify predicators of achievement in an introductory computer science course. SIGMIS Conference '03, April 10-12, Philadelphia, Pennsylvania, USA. 157-161

King, K., and Frick, T. (1999). Transforming education: Case studies in systems thinking. *Adult Education Research Association, 1999 annual meeting*.

Kleinman, J. and Entin, E. (2002). Comparison of in-class and distance-learning students' performance and attitudes in an introductory computer science course. *Journal of Computing Sciences in College*, 17 (6), 206-219.

Ko, A. and Myers, B. (2005). A framework and methodology for studying the causes of software errors in programming systems. *Journal of Visual Languages & Computing*, 16, 41-84

Ko, A. and Myers, B. (2003) Development and evaluation of a model of programming errors. *IEEE Symposia on Human-Centric Computing Languages and Environments,* 2003, Auckland, New Zealand, 7-14.

Koppelman, H., van der Mast, C., van Dijk, E., and van der Veer, G. (2000). The 5th Annual Conference on Innovation and Technology in Computer Science Education, Helsinki, Finland. 97-100.

Lahtinen, E., Ala-Mutka, K., Jarvinen, H. (2005). A study of the difficulties of novice programmers. . The 10th Annual Conference on Innovation and Technology in Computer Science Education, Monre de Caparica, Portugal. 14-18.

Lane, H. and VanLehn, K. (2003). Coached program planning: Dialogue-based support for novice program design. *Proceedings of the SIGCSE symposium on Computer science education*. Reno, Nevada, United States Pages: 148-152

Lehre, R., Guckenberg, T. and Sancilio, L. (1988) Influences of LOGO on children's intellectual development. In R. E. Meyer (Ed.), *Teaching and Learning Computer Programming: Multiple Research Perspectives* (pp. 75-110). Hillsdale, NJ : Lawrence Erlbaum Associates.

Luppicini, R. J. (2002) Toward a conversation system modeling research methodology for studying computer-mediated learning communities. *Journal* *of Distance Education,* 17 (2), 87-101.

Maccia, E., and Maccia G., 1966. *Development of educational theory derived from three theory models.* Washington, DC: U.S. Office of Education, project No. 5-0638. Retrieved October 7, 2005 at http://education.indiana.edu/ ~frick/edutheo.html

Madison, S. and Gifford, J. (2002). Modular programming: Novice misconceptions. *Journal of research on technology in education*, 34 (2), 217-229.

Mancy, R. and Reid, N. (2004) Aspects of cognitive style and programming. *Proceedings of 16th Annual Workshop of the Psychology of Programming Interest Group (PPIG)*, Institute of Technology, Carlow, Ireland, 5-7 April 2004, 1-9.

Marini, A. and Genereux, R. (1995). The challenge of teaching for transfer. In A. McKeough (Ed.), *Teaching for transfer: Fostering generalization in learning*, (pp. 1-20) Mahwah, NJ, USA : Lawrence Erlbaum Associates Inc.

Matlin, M. (1994). *Cognition.* Fort Worth, TX, USA : Harcourt Brace Publishers.

Mayer, R. Dyck, J., Vilberg W. (1986). Learning to program and learning to think: What's the connection? *Communications of the ACM*, 29 (7), 605-610.

Mayer, R. (1985). Learning in complex domains: A cognitive analysis of computer programming. *The psychology of learning and motivation*, 19, 89-130.

Mayer, R. (1988). Introduction to research on teaching and learning computer programming. In R. E. Meyer (Ed.), *Teaching and Learning Computer Programming: Multiple Research Perspectives* (pp. 1-12). Hillsdale, NJ, USA : Lawrence Erlbaum Associates.

McBreen, Pete (2001). *Software Craftsmanship: The new imperative*. New York, NY, USA : Addison-Wesley Professional.

McCraken, M., Almstrum, V., Diaz, D., Guzdial, M., Hagan, D., Kolikant, Y., Laxer, C., Thomas, L., Utting, I., Wilusz, T. (2001). A multi-national, multi-institutional study of assessment of programming skills of first-year CS students. *ACM SIGCSE Bulletin*, 33 (4), 125-180.

McGill, T. and Hobbs, V. (1996). A supplementary package for distance education students studying introductory programming. SIGCSE'96, February 1996, Philadelphia, PA, USA. 73-77.

McGill, T., Volet, S., and Hobbs, V. (1997). Studying programming externally: Who succeeds? *Distance Education*, 18 (2), 236-256.

McKay, E., (2000). Measurement of cognitive performance in computer programming concept acquisition: Interactive effects of visual metaphors and the cognitive style construct. *Journal of applied measurement*, 1 (3), 257-291.

Mento, B, D H Tupper, K Harmeyer, and S Sorkin. Delivering Internet and programming courses online. *ISECON 2000*, Philadelphia, PA, USA.

Merriam, S. (2001). Andragogy and self-directed learning: Pillars of adult education theory. In S. Merriam (Ed.), *The New Update of Adult Learning*

Theory. San Francisco, CA, USA : Josey-Bass.

Mezirow, J. (1991). *Transformative dimensions of adult learning*. San Francisco, CA, USA : Jossey-Bass.

Minsky, M. (2000). A framework for representing knowledge. In J. Haugland (Ed.), *Mind Design II* (pp. 95-128). Cambridge, Massachusetts, USA : Massachusetts Institute of Technology.

Moore, M. and Kearsley, G. (1996). *Distance education: A systems view.* Belmont, CA, USA : Wadsworth Publishing Company:.

Moore, M. and Kearsley, G. (2004). *Distance education: A systems view, 2nd Edition.* Belmont, CA, USA : Wadsworth Publishing Company.

Moore, M. (2001) Standards and learning objects. The American Journal of Distance Education, 15 (3), retrieved October 1, 2005 from http://www.ajde.com/Contents/vol15_3.htm#editorial.

Moore, M. (2004) Research worth publishing. *The American Journal of Distance Education*, 18(3), 127-130.

Morrison, M. and Newman, T. (2001). A study of the impact of student background and preparedness on outcomes in CS1. SIGCSE 2001, Charlotte, NC.

Murphy, L. and Tenenberg, J. (2005). Do computer science students know what they know?: A calibration study of data structure knowledge. The 10th Annual Conference on Innovation and Technology in Computer Science Education, Monre de Caparica, Portugal.

Myers, J., and Munsinger, B. (1996) Cognitive style and achievement in imperative and functional programming language courses. Proceedings of the annual national educational computing conference. USA. 285-293.

Naghsin, R., Seffah, A. , Kline, R. (2003). Cognitive walkthrough+persona = an empirical infrastructure for understanding developers experiences. *IEEE Symposia on Human Centric Computing Languages and Environments (HCC'03)*, Auckland, New Zealand.

Neustadt, R. (1960). *Presidential power, the politics of leadership*. Retrieved March, 2005 at http://www.bartleby.com/73/1514.html

Newell, A. (1990). *Unified theories of cognition*. Cambridge, MA, USA : Harvard University Press.

Noblit G., and Hare. R. (1988) *Meta-ethnography: Synthesizing qualitative studies*. Newbury Park , CA, USA: Sage.

Null, L., Ciaraldi, M., Adams, L., Wolz, U., Hailperin, M. (2002). Rewrite cycles in CS courses: experience reports. 2002 Conference of the ACM Special Interest Group on Computer Science Education, February 27-March 3, Covington, Kentucky, USA. 249-250.

Odekirk-Hash, E. and Zachary, J. (2001). Automated feedback on programs means students need less help from teachers. *Proceedings of the Special Interest Group on Computer Science Education (SIGCSE)*, 02/2001 Charlotte, NC, USA

Odem, H. and Odem, E. (2000). *Modeling for all scales: An introduction to* *system simulation*. San Diego, CA, USA : Academic Press.

Office of vocational and adult education (2004). Community college labor-market responsiveness initiative. Retrieved September 15, 2005 at http://www.ed.gov/ about/offices/list/ovae/pi/hs/factsh/cclmri.html.

Ostby, F. (1999). Improved accuracy in severe storm forecasting by the severe local storms unit during the last 25 years: Then versus now. *Weather and Forecasting, 14(8),* 526-543.

Pallof, R. and Pratt, K. (2003). *The virtual student.* San Francisco, CA, USA : Jossey-Bass.

Parkinson, A., Redmond, J. and Walsh, C. (2004). Accommodating field-dependence: A cross-over study. The 9th Annual Conference on Innovation and Technology in Computer Science Education, Leeds, UK.

Pennington, N. (1985). Cognitive components of expertise in computer programming: A review of the literature. *Psychological documents*, American Psychological Association.

Perkins, D., Schwartz, S., Simmons, R. (1988). Instructional strategies for the problems of novice programmers. In R. Mayer (Ed.), *Teaching and learning computer programming* (pp. 153-178). Hillsdale, NJ, USA : Lawrence Erlbaum Assoc.

Pfleeger, S. and Atlee, J. (2005). *Software Engineering.* New York, NY, USA : Prentice Hall.

Pilke, E. (2004). Flow experiences in information technology use. *International*

Journal of Human-Computer Studies, 61 (3), 347-357.

Pillay, N. (2003). Developing intelligent programming tutors for novice programmers. *Inroads – the SIGCSE Bulletin*, 35 (2), 78-82.

Pirolli, P. (1993) Towards a unified model of learning to program. In E. Lemut, B. du Boulay and G. Dettori (Eds.), Cognitive models and intelligent environments for learning programming (pp. 34-48). New York, NY, USA: Springer.

Prabhakararao, S. (2003) Bringing educational theory to end-user programming. *Proceedings of the 2003 IEEE Symposium on Human Centric Computing Languages and Environments.* Oct. 28-31, 2003 pps 281 – 282.

Prasad, C. & Fielden, K. (2002) Introductory programming: A balanced approach- Proceedings of the 15th Annual NACCQ, Hamiliton, New Zealand, 2002. 101-108.

Pylyshyn, Z. (1984). *Computation and cognition: Toward a foundation for cognitive science*. Cambridge, MA, USA : MIT Press.

Quinn, A. (2002) An interrogative approach to novice programming. *IEEE Symposia on Human Centric Computing Languages and Environments*.

Ramalingam, V., LaBelle, D., Wiedenbeck, S. (2004). Self-efficacy and mental models in learning to program. The 9th Annual Conference on Innovation and Technology in Computer Science Education, Leeds, UK. 171-175

Renaud, K., Barrow, J., le Roux, P. (2001). Teaching programming from a distance: Problems and a proposed solution. *Inroads: SIGCSE Bulletin*, 33

(4), 39-42.

Robbins, J. (1999). Cognitive support features for software development tools. (Doctoral dissertation, University of California, Irvine). Retrieved February 22nd, 2006 at http://argouml.tigris.org/docs/robbins_dissertation.

Romero, P., Lutz, R., Cox, R., du Boulay, B. (2002). Co-ordination of multiple external representations during Java program debugging. *Proceedings of the IEEE 2002 Symposia on Human Centric Computing Languages and Environments*, Arlington, VA, USA. No page numbers.

Rountree, N., Rountree, J., Robins, A. and Hannah, R. (2004). Interacting factors that predict success and failure in a CS1 course. Inroads – the SIGCSE bulletin, 36 (4), 101-104.

Roussev, B. and Rousseva, Y. (2004). Active learning through modeling: introduction to software development in the business curriculum. *Decision Sciences Journal of Innovative Education*, 2 (2), 121-152

Saba, F. & Shearer, R. (1994). Verifying key theoretical concepts in a dynamic model of distance education. *American Journal of Distance Education*, 8 (1), 36-59.

Saba, F. (2003). Distance education theory, methodology and epistemology: A pragmatic paradigm. In M. Moore and W. Anderson (Eds.), *Handbook of Distance Education.* Mahwah, NJ, USA: Lawrence Erlbaum.

Saba, F. (1999) Toward a systems theory of distance education. *The*

American Journal of Distance Education, 13(2), 24-31.

Shaffer, S. (1996). Resurrecting the linguistic relativity hypothesis. Retrieved September 15, 2005 at http://www.scshaffer.com/files/scsshaffer-lrh.pdf

Shaffer, S. (1997, September 29). Try cross-cultural training. *Information Week*.

Shaffer, S. (2004) The uses of systems theory in distance education: An annotated bibliography. DEOS News 13 (7), no page numbers.

Shaffer, S. (2005a). Ludwig: An online programming tutoring and assessment system. *Inroads: The ACM SIGCSE Bulletin*, 34 (2), 56-60.

Shaffer, S. (2005b). System dynamics in distance education and a call to develop a standard model. *International Review of Research in Open and Distance Learning*, 6 (3), no page numbers.

Shaffer, S. (2005c, November). A brief overview of theories of learning to program. *Psychology of programming interest group newsletter*, no page numbers.

Shih, Y. and Slessi, S. (1993). Mental models and transfer of learning in computer programming. Journal of research on computing in education, 26 (2), 154-157.

Sitthiworachart, J. and Joy, M. (2004). Effective peer assessment for learning computer programming. The 9th Annual Conference on Innovation and Technology in Computer Science Education, Leeds, UK. 122-126

Smith, P. and Dillon, C. (1999) *Toward a systems theory of distance education: A response.* The American Journal of Distance Education*, 13*

(2), 32-36.

Sterman, J. (1994). Learning in and about complex systems. *System Dynamics Review*. Vol. 10 (3), 291-330.

Sterman, J. (2000). *Business dynamics: Systems thinking and modeling for a complex world.* Boston, MA, USA : McGraw-Hill/Irwin.

Strauss, D. (2002) The scope and limitations of von Bertalanffy's systems theory. *South African Journal of Philosophy*, 21 (3), 163-179.

Syang, A. and Dale, N. (1993). Computerized adaptive testing in computer science: Assessing student programming abilities. *Proceedings of the twenty-fourth SIGCSE technical symposium on Computer science education*. Indianapolis, Indiana, United States Pages: 53 – 56

Tabs, E. (2003). Distance education at degree-granting post-secondary institutions: 2000-2001. National center for education statistics. Retrieved September 14, 2005 at http://nces.ed.gov/pubsearch/pubsinfo.asp?pubid=2003017.

Tassey, G. (2002). *The economic impact of inadequate infrastructure for software testing.* National Institute of Standards and Technology RTI project number 7007.011.

Thomas, L., Ratcliffe, M. and Robertson, A. (2003). Code warriors and code-a-phobes: A study in attitude and pair programming. SIGCSE'03: Conference of the ACM special interest group on computer science education, Reno, Nevada, USA.

Thomas, L., Ratcliffe, M., Woodbury, J., Jarman, E. (2002). Learning styles and performance in the introductory programming sequence. 2002 Conference of the ACM Special Interest Group on Computer Science Education, February 27-March 3, Covington, Kentucky, USA. 33-37.

Thomas, P. (2002). On the effectiveness of a programming coach in a distance learning environment. ED-MEDIA 2002 World Conference on Educational Multimedia, Hypermedia, and Telecommunications, Denver, Colorado, USA, June 24-29, 2002. no page numbers

Truong, N., Bancroft, P. and Roe, P. (2005). Learning to program through the web. The 10th Annual Conference on Innovation and Technology in Computer Science Education, Monre de Caparica, Portugal.

Tsai, S. (1992). Development of schema knowledge in the classroom: Effects upon problem representation and problem solution of programming. Paper presented at the annual meeting of the American Educational Research Association at San Francisco, August 1992. np.

van Merrienboer, J. (1990). Instructional strategies for teaching computer programming: Interactions with the cognitive style reflection-impulsivity. *Journal of Research on Computing in Education*, 23, 45-53.

van Merrienboer, J., Schuurman, J., de Croock, M. and Paas, F. (2001). Redirecting learner's attention during training: Effects on cognitive load, transfer test performance and training efficiency. Learning and Instruction, 12, 11-37.

von Bertalanffy, L. (1968). *General systems theory: Foundations, development, applications.* New York, NY, USA : George Braziller.

Von Wright, J. (2000). Distance tutorials in a systems design course. The 5th Annual Conference on Innovation and Technology in Computer Science Education, Helsinki, Finland. 105-107.

Vygotsky, L. (1980). *Mind in society: The development of the higher psychological processes.* Cambridge, MA, USA : Harvard University Press.

Vygotsky, L. (1986). *Thought and language.* Cambridge, MA, USA : The MIT Press.

Watzlawick, P. (1984). An introduction to radical constructivism. In P. Watzlawick (Ed.), *The invented reality.* New York, NY, USA: W. W. Norton and Company.

Williams, L., Kessler, R., Cunningham, W. and Jeffries, R. (2000). Strengthening the case for pair-programming. *IEEE Software*, 17 (4), 19-25.

Woit, D., and Mason, D. (2003). Effectiveness of online assessment. *Proceedings of the SIGCSE symposium on Computer science education*. Reno, Nevada, United States Pages: 137-141.

Zachary, J. and Jensen, P. (2003). Exploiting value-added content in an online course: Introducing programming concepts via HTML and JavaScript. *Proceedings of the Special Interest Group on Computer Science Education (SIGCSE)*, Feb 19-23, Reno Nevada, USA.

Zachary, W., Ryder, J. and Hicinbothom, J. (2000). Cognitive task analysis and modeling of decision making in complex environments. In J. Cannon-Bowers and E. Salas (Eds.). *Making Decisions Under Stress : Implications for Individual and Team Training* (pp. 315-344). Washington, D.C., USA : American Psychological Association.

Zeller, A. (2000). Making students read and review code. *Proceedings of the 5th annual SIGCSE/SIGCUE ITiCSE conference on Innovation and technology in computer science education*, Volume 32 Issue 3 , July 2000, Helsinki, Finland.

Appendix A – Quotations on variable descriptions from the literature

1. McKay (2000): “Having prior knowledge is not adequate on it own; it must become activated in a semantic context” (p. 259).
2. Perkins (1988): "that often students had a knowledge of relevant command structures, but could not retrieve them, apparently failing to make the connection between what needed to happen in the program and the commands that would serve the purpose" (p. 165).
3. Madison & Gifford (2002): “Researchers studying novice programmers have found that most faulty answers … arise from a systematic application of knowledge the student already has” (p. 218).
4. Madison & Gifford (2002): “Teachers who view learning … within a constructivist framework should not be surprised that students construct their own meanings. More importantly, they should not be surprised that some students harbor misconceptions despite instruction to the contrary” (p. 219).
5. Arzarello, Chiappini, Lemut, Marara, and Pellery (1993): “(i) there is no everyday intellectual activity that can form the basis for spontaneous constructions of mental models for programming concepts; (ii) the physical machine is not cognitively transparent” (p. 289).
6. Blackwell (2002): In geometry, "many constraints on causality are made directly available via the user's perception of the apparently

physical situation” (np).

7. Blackwell (2002): In programming “there is no limit on the abstract expressive power of the representational system, and hence no boundary that can be exploited to constrain reasoning during planning” (np).
8. Arzarello, Chiappini, Lemut, Marara, and Pellery (1993): “Conflict emerges when the novice must get away from natural language or another familiar one to express himself in a new artificial language, in which it is not possible to translate in a one-to-one fashion” (p. 289).
9. Blackwell (2002): A program is “an abstract notation defining required behavior in different circumstances – a conflict with the ideal of direct manipulation. Of course these departures from the 'ideal' are not a bad thing -- they are necessary in order to achieve the task" (np).
10. Holden and Weeden (2004): “The hurdle in programming appears to be learning the basic concepts such as sequence, iteration, and decision. Once these are learned, students are able to master the more advanced concepts of later courses” (p. 217).
11. Arzarello, Chiappini, Lemut, Marara, and Pellery (1993): “Experts are able to construct abstract representations of the program aims, while novices usually realize more concrete representations of its operations” (p. 286).
12. Ramalingam, LaBelle and Wiedenbeck (2004): “Programming is a

highly cognitive activity that requires the programmer to develop abstract representations of a process in the form of logic structures. Having a well-developed and accurate mental model may affect the success of a novice programmer in an introductory programming course. Such a model could include knowledge about how programs work in general, as well as knowledge about the syntax and semantics of a specific language. Mental models … play an important role in programming comprehension and correspondingly in comprehension-related tasks, such as modification and debugging" (p. 172).

13. Arzarello, Chiappini, Lemut, Marara, and Pellery (1993): "The interpretation [of a program] is constantly mediated by the student's knowledge of the problem domain and by the quality and number of programming schemes he knows" (p. 285).
14. Blackwell (2002): One of the things which makes programming difficult is that it requires two different types of abstractions: "abstraction over time" vs. "abstraction over a class of situations" (np).
15. Pirolli, P. (1993): "We assume that when students are faced with some novel programming situation, they use their conceptual understanding of programs (and how to write them) to explicitly reason about possible solutions… (but) when successful, a constructed mental model leads to the proposal of a programming operator that solves the particular coding impasse. Chunking will result in new productions that

will correctly propose and apply the operator in future similar situations without the need for consulting the mental model. In this way, declarative knowledge of programming becomes situation-specific procedural knowledge" (p. 38).

16. Arzarello, Chiappini, Lemut, Marara, and Pellery (1993): “As many researchers have observed, the notion of the machine in computer environments is far from being transparent. It differs greatly from the notion of everyday life machines. In fact, A. Turing invented programming machines exploiting the new concept of *internal states*; the comparison between a sausage machine and a computer shows that both are devices with an input and an output, but the latter can only be described completely by using the very abstract notion of internal states. It is a very hard notion to assimilate, as many researches have proved. In fact, the knowledge used by an expert in programming extends beyond operating rules and involves the representation of the whole system” (p. 291).
17. van Merrienboer, Schuurman, de Croock, and Paas (2001): “The acquisition of complex cognitive skills is heavily constrained by the limited processing capacity of the human mind. Cognitive load theory provides guidelines to circumvent those limitations in training situations. It provides guidelines to: (1) prevent cognitive overload; (2) decrease extraneous cognitive load which is not relevant to learning;

and (3) increase, within the limits of total available cognitive capacity, germane cognitive load which is directly relevant to learning" (p. 12).

18. Ko and Myers 2005: "Because knowledge-based activities rely heavily on the interpretation and evaluation of models of the world (in programming, models of a program's semantics), they are considerably taxing on the limited resources of working memory" (p. 54).
19. van Merrienboer, Schuurman, de Croock, and Paas (2001): "Training with completion problems or worked examples required the same amount of mental effort and led to higher transfer test performance, combined with lower cognitive load during the test than training with conventional problems" (p. 13).
20. Robbins, 1999 : "Mental context switches occur when designers change from one task to another. When starting a new step or revisiting a former one, designers must recall schemas and information needed for the task that were not kept in mind during the immediately preceding task" (np).
21. Foreman, 1990: "The difficult nature of programming and the fact that the cognitive styles of adults are well-established may explain the wide variety of ways in which adults approach programming" (p. 55).
22. McKay, 2000: "When the intellectual task places extraneous cognitive load on a novice learner there is an interactive effect of cognitive style

and instructional format on instructional outcome. Consequently, individualized instructional strategies which are designed to respond to the differences in [cognitive style], reduce the burden on poor working memory" (p. 259).

23. McKay, 2000: Presents a detailed statistical study (n=194) from which he concludes that "there is most likely an interactive effect of cognitive style and type of instructional treatment on cognitive performance" (p. 278), specifically with regard to students learning computer concepts.
24. McKay, 2000: "Picking out these important instructional variables (spatial ability, and method of delivery) for some types of instructional outcomes progresses our ability to provide instructional environments for a broader range of novice learners, thereby giving them a choice of information-agent, controlling the choice of instructional format and instructional event conditions" (p. 280).
25. Foreman (1990): "It might be that a link between the cognitive requirements needed for programming and individual cognitive style could be enhanced by instruction that facilitates cognitive restructuring" (p. 59).
26. van Merrienboer (1990): "a preferential model seems to be more adequate than a compensatory model to describe the relations between instructional strategies and reflection-impulsivity" (np).
27. Heo (2003): "in larger and/or distributed classes… the instructor does

not always have the luxury of devoting extensive time to assessing individual learning styles. Thus, these courses are often taught in a passive environment with generalized lecture material and coding assignments are often graded with limited feedback in problem solving and programming techniques" (p. 151).

28. Pirolli, 1993: "Analysis suggests that the initial acquisition of cognitive skills in programming is highly dependent on the relevant declarative knowledge that is available. Indeed, protocol studies, production system simulations, and pedagogical experiments show that declarative conceptual models of instructional texts and example programs have a significant impact on the acquisition of programming skill" (p. 40).
29. Parkinson, Redmond and Walsh, 2004: "Studies suggest that individuals who are Field-independent perform better than those who are Field-dependent in computer-assisted and hypermedia-assisted environments" (p. 72).
30. Parkinson, Redmond and Walsh, 2004: "Field dependent individuals will tend to re-organize, restructure or represent information to suit their own needs, conceptions or perceptions" (p. 72)
31. Bishop-Clark (1995): "Designing the solution for a computer program requires the programmer to identify relevant parts of a problem and structure the parts in ways that will solve the problem at hand. In

other words, the task of designing solutions requires programmers to impose structure (for the computer) on a relatively unstructured environment (the real world). It is a task that requires students to use the restructuring skills like those described in field-independent students" (p. 375).

32. Davies (1994): "It is suggested that schemata structures may be viewed as hierarchically structured propositional representations. The salient elements of programming plans, that is those elements that encode information relating to the current goal (focal lines), will occur at higher levels of the propositional hierarchy. Such a form of representation may go some way towards capturing the structural organization of programming knowledge" (p. 713).
33. Madison & Gifford, 2002: "Because new knowledge builds recursively on existing knowledge in this model, each student constructs an idiosyncratic version of the knowledge" (p. 218).
34. Davies, 1994: "The specific learning or training experience of the programmer contributes significantly to the development of hierarchical representations of programming knowledge" (p. 713).
35. Foreman (1990): "It is believed that structural programming practices put less information processing burdens on students because they are dealing with a chunk of information at a time. Therefore, instructors may emphasize top-down design and program modularity by

encouraging students to construct problem solutions away from the computer. Diagrams or flowcharts could also be used to help students organize and refine problem solutions" (p. 60).

36. Roussev, B. and Rousseva, Y., 2004: "Knowledge learners tend to resort to less conclusive and less constrained abductive and adductive processes" (p. 126).
37. Foreman, 1990: "The ability to reason inductively proved to be an important factor in the initial acquisition of program comprehension, composition, debugging and modification skills" (p. 59)
38. Tsai, 1992: "inductive reasoning ability has been shown to influence programming achievements" (p. 7).
39. Pirolli, 1993: We want to determine "the effects of pedagogical manipulations that influence the initial acquisition of programming operators. For instance, it is possible to determine which programming operators in a learning situation can be induced by analogy to example programs presented in instruction" (p. 40).
40. van Merrienboer, Schuurman, de Croock, and Paas , 2001: "Superior cognitive schemata [are] constructed in the completion and worked example conditions" as opposed to the conventional problem solving approach wherein the student must create a program "from scratch" (p. 14).
41. McKay, 2000: "Several researchers have documented the ability of

the metaphor to trigger prior domain knowledge in the acquisition of new concepts” (p. 259).

42. Roussev and Rousseva, 2004: “Since modeling is based on the mechanics of metaphor application, modeling evokes the same cognitive process as metaphor, and therefore creates a condition for active learning… Since modeling employs abstraction, modeling has the ability to deepen our understanding of the designed system by reducing the context, which the designer needs to comprehend, to the most essential system properties" (p. 122).

43. McGill and Hobbs, 1996: “Explicit instruction in program design strategies has a beneficial effect on student’s learning, and that the best results are obtained when teachers model their own design processes at a level accessible to students … An instructional strategy emphasizing expert demonstration and interactive guided practice has been successfully piloted. Students taught using this strategy developed a better understanding of programming principles and demonstrated a greater ability to apply their programming knowledge to novel problems” (p. 73).

44. Perkins, 1988: "Research in programming has shown that many errors made by novice programmers can be attributed to a tendency to understand the computer as a personlike entity that comprehends intensions" (p. 164).

45. Arzarello, Chiappini, Lemut, Marara, and Pellery, 1993: “The inability of the computer to interpret the student’s intentions … and the need to overcome this rigidity of the computer in order to realize executable programs, strongly moves the novice’s attention from the semantics of the problem to the syntax of its implementation, and in particular to the syntax of a single program line” (p. 290).
46. McGill and Hobbs, 1996: Whereas students have difficulty constructing their own patterns (also called templates), “studies suggest that representing programming knowledge in the form of short templates can help students write programs” (p. 73).
47. Capretz (2003): “Software engineers and psychological types are clearly related ... more specifically, the current work suggests that software engineers are most likely to be STs or TJs or NTs. The results are important to employers looking for software professionals and to students looking for careers … although interests and personality types may play a role in the selection of a career, they may not predict success in that area ... Due to the diverse nature of software engineering, it is widely believed that no personality instruments will ever accurately predict success in this field” (p. 212). (Note that “ST” is a “sensing/thinking” person, a “TJ” is a “thinking/judging” person, and an “NT” is an “intuitive/thinking” person.)
48. Foreman, 1990: “Since learning to program is a new experience for

most students seeking computer literacy, fluid ability may be more closed related to computer programming success" (p. 56) than ability based on previous knowledge.

49. Foreman, 1990: "The ability to follow a sequence and attend to multiple variables seems to be an important factor for the acquisition of programming skills" (p. 60).

50. Pirolli, 1993: "In general, it appears that students prefer to construct mental models of example programs and then elaborate these models using propositions from the text rather than vice-versa." (p. 43).

51. Bruce, 2004: "Most faculty fall into the minority of the population that seems to learn best via abstractions and principles first, followed by concrete examples" (p. 32).

52. Lahtinen, Ala-Mutka, Jarvinen, 2005: "The biggest problem of novice programmers does not seem to be the understanding of basic concepts but rather learning to apply them" (p. 17).

53. Madison and Giffford, 2002: "Traditional instruction has been designed to develop skills, perhaps at the expense of understanding" (p. 227).

54. Madison and Gifford, 2002: "A number of studies have shown a correlation between programming success and background factors such as student demographic information or cognitive style; however, this information is not very helpful to programming teachers. Though

background factors may assist advisers counseling students regarding the selection of programming classes, teachers cannot alter their students' age gender, verbal or mathematical ability, or high school rank, for example, and thereby improve student success in programming" (p. 217).

55. Katz, Aronis, Allbritton, Wilson and Soffa, 2003: "An instrument to identify students early on who are likely to succeed (or not) in introductory courses that focus on programming concepts and skills. This information could be used by instructors to determine which students are likely to need extra help" (p. 161).
56. Arzarello, Chiappini, Lemut, Marara, and Pellery, 1993: Top-down refinement requires "a nontrivial ability to plan, which implies the use of anticipating thought aimed at determining a hierarchy of goals related to the problem representation that the subject constructs dynamically" (p. 286).
57. McGill and Hobbs, 1996: There is "evidence that expert programmers spend much of their programming time designing and planning problems solutions before coding the program into a specific language" (p. 73).
58. Arzarello, Chiappini, Lemut, Marara, and Pellery, 1993: "The expert can model and scaffold the novice, and the novice can observe the expert, correct and refine his own practice looking carefully at the

expert’s externalization of his relevant internal cognitive processes and methods. The novice must observe, analyze, and evaluate not only inner cognitive processes, which the expert externalizes, but he must also compare and contrast them with his own intellectual performance. A sort of continuous self-evaluation is required; the point is that the novice can activate it, only if his distance from expert performances is not too far; if not, this may cause stumbling blocks in cognitive apprenticeship” (p. 292).

59. van Merrienboer, Schuurman, de Croock, and Paas, 2001: “A mindful abstraction … [is] … a controlled, effort demanding process [that] will therefore increase cognitive load” (p. 14).

60. Tsai, 1992: "The degree of students' mindfulness as a tendency influences their learning" (p. 7).

61. McCraken, Almstrum, Diaz, Guzdial, Hagan, Kolikant, Laxer, Thomas, Utting and Wilusz, 2001: Found what they called “troubling” bi-modality: “This result suggests that our current teaching approach is leading to one kind of performance for one sizable group of students and another kind of performance for another sizable group” (p. 132).

62. Hudak, M. and Anderson, D., 1990: The latter Piagetian stage "entails grasping the logic of all possible combinations of items or events, developing a combinatorial system, and unifying operations into a structured whole... data indicate that as many as 50% or more of

college students are not formal operators.. [some] express alarm, arguing that the use of nonformal operational thinking by college students is increasing, and propose methods for promoting stage advancement" (p. 231).

63. Hudak and Anderson, 1990: “These findings emphasize the need to examine students' cognitive maturity and learning style – factors often ignored in research aimed at ascertaining the reasons for academic success at the college level. [These] results have important implications for theoretical understanding of the problem and for potential remediation" (p. 233).
64. Hudak and Anderson, 1990: “Findings highlight the need to examine both cognitive maturity and learning style in studies of academic success at the college level" (p. 231).
65. Marini and Genereux, 1995: “Broadly defined, transfer involves prior learning affecting new learning or performance. The new learning or performance can differ from original learning in terms of the tasks involved (as when students apply what they have learned on practice problems to solving a new problem), and/or the context involved (as when students apply their classroom learning to performing tasks at home or at work)” (p.2).
66. Roussev and Rousseva (2004): proactive interference is “the activation of an inappropriate schema that can interfere with both the

initial comprehension and subsequent retrieval of information. The result of the proactive interference is a more difficult learning process than if there were no prior knowledge" (p. 125).

67. Arzarello, Chiappini, Lemut, Marara, and Pellery, 1993: “It is not easy for the novice to dissociate the ‘while’ [command in programming] … from the ‘natural’ meaning of temporal continuity to discretize them” (p. 291).
68. Morrison and Newman, 2001: “We were surprised to find that those students who took only a BASIC programming course prior to taking CSI did significantly worse than those students who took a prior course in some other programming language” (p. 182).
69. van Merrienboer, Schuurman, de Croock, and Paas, 2001: “High contextual interference yields better retention and higher transfer of acquired skills than low contextual interference, but that learners who practice under interfering conditions will typically need more time and invest more mental effort to master the skills” (p. 14).
70. Anderson (2004b): “No element of course design concerns the student in a formal educational context more than that related to assessment. Effective teaching presence demands explicit and detailed discussion of the criteria on which student learning will be assessed” (p. 281).
71. McCraken, Almstrum, Diaz, Guzdial, Hagan, Kolikant, Laxer, Thomas, Utting and Wilusz , 2001: Multiple choice exams of LP are invalid

"because of the limited time available for students to complete them, but they can provide cognitive complexity at a low cost" p. 127).

72. McCraken, Almstrum, Diaz, Guzdial, Hagan, Kolikant, Laxer, Thomas, Utting and Wilusz, 2001: "Multiple-choice questions cannot directly test students' ability to create working computer programs" (p. 127).
73. Moore and Kearsely (2004): "The assignment is the key component that links the instructor to the student … and even the student to other students" (p. 138).
74. Anderson (2004b): "We know, from research on assessment, that timely, detailed feedback provided as near as possible to the performance of the assessed behavior is most effective in providing motivation and in shaping behavior and mental constructs" (p. 281).
75. Cheang, Kurnia, Lim and Oon (2003): "Programming assignments are notoriously difficult to manually grade in a fair and timely manner ... in fact, the labor-intensiveness of grading programming assignments is the main reason why few such assignments are given to the students each semester, when ideally they should be given more" (p. 122).
76. Anderson (2004b): "Machine evaluations, such as those provided in online multiple-choice test questions or in simulations, can be very effective learning devices" (p. 281).
77. Thomas (2002): The coach is "a software component that can be invoked on demand to provide a variety of support based on students'

previous experiences” (np).

78. Buck & Stucki (2000): “For some students, whose cognitive development is already advanced, this may be appropriate. However, with the increasing diversity of backgrounds of students selecting computer science as a major, we no longer have the luxury of ignoring pedagogical issues” (p. 76).

79. Anderson and Skwareki, 1986: Automated tutoring systems “serve as tools for collecting data about programming behavior and for producing experimental manipulations. Given that the effectiveness of these tutors is predicated on our understanding the cognitive processes involved in programming, we expect our research with them to lead to improved tutoring based on a deeper understanding of programming” (p. 849).

80. Pillay (2003): “Presenting and explaining programming concepts; providing students with different types of programming problems to work through. Each type of problem will test a different aspect of programming; assisting students to develop solutions to the different types of programming problems and assessing student solutions; assessing programs written by students with respect to correctness and efficiency; assisting students in debugging their programs for semantic errors. This basically involves detecting logical errors, identifying student misunderstandings, explaining these to the

students and advising the student of how to fix programming bugs" (p. 79).

81. Anderson and Skwareki (1986): The tutor reacts to each symbol as the student types it. A student cannot create and edit a sequence of symbols before getting feedback from the tutor on any of the symbols." (p. 846). They propose that "immediate feedback is usually best" (p. 844) and that "delayed feedback proves difficult in terms of pedagogical effectiveness" (p. 846).
82. Holden and Weeden (2004): "One programming language use in prior experience does not seem to indicate future success more than others. This also would support that learning the basic concepts is the hurdle" (p. 217).
83. Prasad and Fielden, 2002: "It should be noted that the choice of a teaching language is usually dependent on current market trends and suitability to teaching" (p. 104).
84. Prasad and Fielden, 2002: "The argument still remains that programming languages are merely tools to illustrate the concepts of programming, and that a good programmer should be able to adapt to any programming language despite the language that s/he was taught in" (p. 104).
85. Lahtinen, Ala-Mutka, Jarvinen (2005): "If small examples, emphasizing few concepts at a time, could be developed to support

students' active programming skills, they would also better engage the student in the learning situation" (p. 17).

86. Heo, 2003: "Currently, in most distributed education environments, student assignments are uploaded to the course server or delivered to the teacher by email. The instructor then reviews the submitted assignment and adds comments at the end of the assignment file or in a separate email message which is returned to the student. This practice creates added difficulty since, in addition to evaluating and commenting on the code, the instructor must now consider the line number and location of the applicable comment. This also presents added difficulty for students since they must orient themselves to the specific location of comments by counting line numbers and attempting to apply consolidated comments to appropriate sections of the code" (p. 153).

87. Madison and Gifford, 2002: "Programming teachers should refrain from interpreting program errors as simple carelessness or lack of attention or concern on the student's part" (p. 226).

88. Roussev, B. and Rousseva, Y., 2004: "Empirical results show that retrieval of an appropriate source is the bottleneck in analogical reasoning and often can only be performed successfully if explicit hints about a suitable source are given" (p. 127).

89. Anderson, 2004b: "First and primarily, an excellent e-teacher is an

excellent teacher. They like dealing with learners; they have sufficient knowledge of their subject domain; they can convey enthusiasm both for the subject and for their task as a learning motivator; and they are equipped with a pedagogical (or androgogical) understanding of the learning process, and have a set of learning activities at their disposal by which to orchestrate, motivate, and assess effective learning" (p. 290).

90. Anderson (2004b): "In fulfillment of this component of teaching presence, the teacher regularly reads and responds to student contributions and concerns, constantly searching for ways to support understanding in the individual student and the development of the learning community as a whole" (p. 280).

91. Clancy, Titterton, Ryan, Slotta and Linn (2003) propose a lab-based model for LP/F2F where "the role of the instructor had changed fundamentally ... Instead of being primarily a lecturer, the instructor becomes a tutor, spending most of his time engaging students one-on-one... [using automated tools, the instructor] can more efficiently target the tutoring attention where it is necessary" (p. 135).

92. Arzarello, Chiappini, Lemut, Marara, and Pellery (1993): "The art of making *cognitive* the *apprenticeship* mainly consists in the creation of a balanced dialectic between scaffolding and challenging the knowledge of the novices, handling the conflicts which emerge. The

expert must clarify which are the points where there is contradiction between old and new knowledge and must propose problems to novices, which allow them to recognize conflicts, as a first step towards overcoming them. Apprenticeship becomes really cognitive in that the expert helps novices in modifying, re-organizing and developing their knowledge" (p. 292).

93. von Wright (2000): "this kind of distance teaching is well suited for computer science courses where the aim is to learn a method and an associated design notation" (p. 107).
94. Anderson, 2004b: "Discourse … helps students uncover misconceptions in their own thinking, or disagreements with the teacher or other students. Such conflict provides opportunity for exposure of cognitive dissonance that, from a Piagetian perspective, is critical to intellectual growth" (p. 280).
95. Hulkko & Abrahamsson (2005): "Pair programming may not necessarily provide as extensive quality benefits as suggested in the literature, and on the other hand, does not result in consistently superior productivity when compared to solo programming" (p. 495).
96. Clancy, Titterton, Ryan, Slotta and Linn, 2003: "Instead of listening to lectures, students would be participating in computer-based activities (e.g., programming exercises) and instructors would be freed up to interact more closely with students as they worked in pairs or small

groups" (np).

97. Buck & Stucki (2000): "In computer science, an error has been made by assuming that the student should start out by writing the equivalent of [an entire essay]" (p. 75).

98. Anderson, 2004b: "Three critical roles that a teacher performs in the process of creating an effective teaching presence. The first of these roles is the design and organization of the learning experience that takes place before the establishment of the learning community and during its operation. Second, teaching involves devising and implementing activities to encourage discourse between and among students, between the teacher and the student, and between individual students and groups of students and content resources. Third, the teaching role goes beyond that of moderating the learning experiences when the teacher adds subject matter expertise through a variety of forms of direct instruction " (p. 274).

99. (Blackwell, 2000, np): "The Attention Investment model is a decision-theoretic account of programming behavior. It offers a cost/benefit analysis of abstraction use that allows us to predict the circumstances in which users will choose to engage in programming activities, as well as helping tool designers to facilitate users' investment decisions and reduce the risks associated with those decisions… The Attention Investment model successfully simulates a range of observed user

actions, accounting for them on the basis of previous experience, rather than assuming that different classes of user are either capable or incapable of programming because of their intellectual abilities. Removing this apparent discontinuity in the user population provides the opportunity to apply a single cognitive model of programming processes that can apply both to end-users and to everyday decisions made by professional programmers".

100. Shih and Alessi, 1993: “Instruction that emphasizes ‘how to’ can be effective in a particular context but may not transfer to novel situations because it does not teach the knowledge underlying the skills. On the other hand, instruction that emphasizes the ‘why’ can provide richer knowledge applicable to a variety of contexts but creates a discrepancy between instruction and application – that which we teach is not what we expect students to do” (np).

101. Madison and Gifford, 2002: “A number of studies have shown a correlation between programming success and background factors such as student demographic information or cognitive style; however, this information is not very helpful to programming teachers. Though background factors may assist advisers counseling students regarding the selection of programming classes, teachers cannot alter their students’ age, gender, verbal or mathematical ability, or high school rank, for example, and thereby improve student success in

programming" (p. 217).

102. Califf and Goodwin (2002): "Since we believed that a number of students were continuing past the first programming course without having acquired the skills necessary to write a working computer program on their own and that many did not even see this as something they should be able to do, we decided to institute a laboratory final examination that would require students to demonstrate the ability to write very simple computer programs without aid. The goals of the exam are fairly simple: to ascertain if the student is able write a program from scratch, compile it, find any errors, and ultimately have the program solve the problem" (p. 218).

103. Moore & Kearsley, 2005: "Ideally there will be a subsystem for scanning the social environment (some people would call it a market), and for making the determination of what to teach on the basis of data about needs and demand. This includes finding out what knowledge students themselves feel they need" (p. 13).

104. Renaud, Barrow and le Roux (2001): "We do not want to become merely a service industry which delivers the skills that industry requires at any given time, but rather strive to produce graduates who have learnt how to think and who can develop the skills required by industry. There needs to be a balance between assimilation of long-term concepts and the learning of short-term skills. A long-term

knowledge base ensures that students are adaptable to current organizational needs” (p. 41).

105. Pallof and Pratt (2003): “Instead of being the ‘softer, easier’ way, online courses are actually estimated to require at least twice as much time as regular classes” (p. 77) .

106. Holden and Weeden (2004): “There was a significant increase in performance among students who had prior programming experience, but there were decreasing marginal returns from increasing levels of prior experience. Students with minimal experience earned almost a full letter grade (0.89) higher than the students with no experience, while there was only a 0.31 grade increase from minimal experience to medium experience and only 0.24 from medium to very experienced. In all cases, students with any level of experience exceeded the average for all students (2.49)” (p. 214).

107. Buck & Stucki (2000) : “Because they have just learned algebra, and the symbology is similar, they somehow think that when they type in what appears to be an equation that the computer is going to solve it” (p. 79).

108. Emory and Tamassia (2002): even writing “the simplest program requires knowledge of the inner workings of the [programming] language environment ... typically these details, though important to the successful compilation and execution of ... programs, are not the

focus of introductory computer science courses ... [Our] goal is to teach the principles of ... programming without overwhelming the students with the technical idiosyncrasies of [the environment]" (p. 307).

109. Hasker (2002): "CS1 students, especially those without previous programming experience, struggle with a host of difficult issues including formal syntax, algorithmic problem solving, and general computer science survival skills. Such students should not have to struggle with the tools as well" (p. 56).

110. Truong, Bancroft and Roe, 2005: "Unfortunately, when attempting to write their first programs, novices may be hindered by difficult side issues such as learning how to use a program text editor, installing a language compiler, and knowing how to compile and debug the program using the error messages generated by the compiler" (p. 9).

Appendix B – Biographies of student participants

Students were selected for the qualitative study to obtain results from a wide diversity of backgrounds and life situations. This section presents short biographies as submitted by the students near the beginning of the study. Details that might identify the student have been removed from the text.

Subject 1. "I am a 47 year old woman who is married and has been married for 22 years living in our own home. I have one daughter who is now in college. I have one living parent. My father, age 79, lives in Michigan in his own home. I have one living in-law, my mother-in-law, who lives in a retirement community nearby. I recently returned to the workforce part-time in an office environment where I am brushing up on my Microsoft Office skills. I have a BA degree in psychology in 1979 from Alma College, Alma Michigan. I took 2 programming classes in college. I found that made me very marketable and I was in demand for programming jobs of that time. I worked 8 years at various programming positions until the birth of my daughter in 1986."

Subject 2. "I'm 19, and live in a suburb outside of [city] called ________. I attended _______ High school, and graduated in 05. I am currently attending [this community college] because of its low cost, and it's pretty close to my home (about 20 minute drive). I took this online class because I could do it from my house, and I enjoy using the computer, although I'm finding it more challenging than I had expected. I like playing video games on my spare time, x-box and online. I play with my friends and we use voice chat programs, such as Ventrillo or Teamspeak. My dad died when I was two, my mom remarried when I was about 7, but they are currently separated. My step dad lives in _______, Pennsylvania, and I see him occasionally. I live with my mom here in _______. I have one older brother who is 26, and lives in the area as well. I currently work at ________. I usually wait on customers, and clean up the place."

Subject 3. "I am a 24 year old college student. I graduated high school in 1999 and attended community college for a year. After that I attended [a local university] for a couple years, but dropped out because I couldn't concentrate on attending classes and doing my schoolwork while working full-time and worrying about the health of both parents, who went through a couple of close calls. My parents are both healthier now and I want to get a better job so I decided it was time to return to school. I am employed part-time at a ______ convenience store. I worked full-time until the beginning of the semester, but once I gave my manager my new availability my hours got cut drastically and I'm trying to get them back up to at least 30 a week, so I can afford food, gas, etc. I live with my boyfriend who moved in with his mother when she bought a house because she couldn't afford it on her own. She recently got remarried but my boyfriend and I still live in his basement apartment because we can't afford to move out on our own while I'm still in school and he's in his apprenticeship."

Subject 4. "I am 33 Years Old. I went to school in India before this and had my Bachelors Degree in English from India. I am working in a PM Engineer in an

Pharmaceutical Company dealing with Equipment and Work orders and Preventive Maintenance on Equipment. I am living with my spouse and my two children."

Subject 5. "I am a 38 year old married woman with two college aged children. I have been married 20 years and my 20 year old daughter goes to [this community college] as well and my 18 year old son goes to [a local four-year college]. I currently live with my husband and daughter, my son lives on campus but will return home for summer. I graduated from _______ HS in 1985 and married two months later. I am returning to school for the first time since then. I work at _______ as a technical support person."

Subject 6. "Well I'm 19 years old, I went to _______ High School in _______, I was a student there for a little more than two years and graduated in 2004, before that I was at _______ in _______ NJ. I babysit two little children sometimes before school or after school a few hours each week, when I'm not there I work as an order taker at Wendy's. They're not my ideal jobs but they fit real well with my schedule of balancing my PA social life as well as my NJ social live (I was there for about 11 years, essentially grew up there and since it is only about an hour and half away I still go back often) in addition to school as well. I used to skate in my free time, play volleyball and lacrosse at school but due to a knee injury my speed slowed me down and I stopped for a while, trying out other things instead of devoting all my free time to sports as I used to. Due to having a vast amount of interests and trying to control my spending by opting out of loans, I decided to commute to college and the second closest one was _______, where I am currently a sophomore student. I live with both my parents and a younger brother."

Subject 7. "My name is ________ and I am 18 years old. I just graduated from ________ Highschool in 2005 and I am currently taking classes at the ___________. I currently am working part time for _______. I am a shift leader there and would like to move up to assistant manager in the near future. I've been working for [that company] since I turned 17 and plan on staying with the company until I am out of college. Since my work schedule changes every week I decided to take all of my classes at _______ over the internet so that I can do my school work anytime I want. I am currently living at home with my parents. I would like to move out on my own but can not afford it at this time. I enjoy hanging out with my friends as much as possible. I surf the internet and talk to my friends over AOL Instant Messenger at least 5 hours a day. My computer is almost always on, the only time I shut it off is when I know I am going to stay overnight some place. I am a late night person and stay up till at least 2 am every night. That is also when I do 90% of my school work."

Subject 8. "I am 24 years old, and I grew up in the _______ area all of my life. I attended _______ High School and waited a while to attend college after because I was so confused about what I should go to college for. I didn't want to waste my money on something I might not want to do down the road. I met my husband, _______ when I was in my senior year of high school and because of my parents having difficulty with their marriage, I moved in with my grandmother. I then moved out and moved in with _______ and started to try and get my life together. I started college, bought a house with _______, got married, sold that house and moved out, and just recently, purchased a house in the country where my husband and I are finally going to say put. I could not get grant money when I started college because the Board of Education would only consider my parents income even though I had no help from them and was living on my own. That was very hard to hear and upsetting because I did not have the money, however, I finally became close with my mom and attained a scholarship through where my mom had worked. I had to attend college full-time to keep my scholarship which made it hard financially because I couldn't handle working full-time at the same time, and I needed to help pay for my house. Recently, my scholarship has ended so I am going to be attending college part-time and I am currently trying to find a full-time job with benefits. I haven't had health insurance in so long, and really need to the benefits as well as the money. I am currently working part-time for a company called ___________________ as an office manager/project secretary."

Steven C. Shaffer – Curricula Vitae

Education and training

M.S., Computer Science, Villanova University, 1990
B.A., Computer Science, Temple University, *Summa Cum Laude*
Certified Software Quality Engineer, American Society for Quality, 1998
Certified Systems Professional, ICCP, 1987

Academics positions

Instructor, Computer Science & Engineering, The Pennsylvania State University, 1999 – present. Teaching Intermediate Computer Programming, Software Engineering, Artificial Intelligence; Developed a computer science – specific on-line testing system; developed and taught an eight-course sequence in Internet Software Development for the Department of Continuing Education, member of the curriculum committee. Outstanding teacher award, 2004
Instructor, Computer Programming, South Hills School of Business & Technology, 1999-2000.
Adjunct Professor of Computer Science, Ursinus College, 1990-1996.
Instructor, Computer Science, Villanova University, 1991.
Instructor, Computer Programming, Lansdale Junior College, 1986-1988.

Industry positions

President / Consultant, Systemics Inc., 1998 – 2002.
President / Senior Consultant, Decision Associates, Inc., 1993 – 1997.
Senior Consultant, Integrated Systems Consulting Group, 1989 – 1993.
Contract Software Engineer, various organizations, 1986-1993.
Senior Programmer, Control Data Corporation, 1988 – 1989.
Systems Analyst, Analytics Inc. 1986 – 1988.
Software Developer, Softerware, Inc., 1984 – 1986.

Publications

System Dynamics in Distance Education and a Call to Develop a Standard Model. International Review of Research in Open and Distance Learning November, 2005
A brief overview of theories of learning to program. Psychology of programming interest group -- newsletter October, 2005
Ludwig: An online programming tutoring and assessment system. Inroads: The official quarterly publication of the ACM special interest group on computer science education. June, 2005.
The uses of systems theory in distance education. Distance Education Online Symposium, September, 2004.
Putting the self back in the social: rescuing cognitivism from incautious rhetoric. Adult Education Research Conference, Vancouver, BC May, 2004.
An Algorithm for Comparing Labeled Graphs, Learning Technology, vol 6, no. 3, July 2004.
Try cross-cultural training. Information Week, September 29, 1997.
The effects of different types of strategy training on the reading performance of young and old adults. (co-author) Report presented at the Gerontology Center in Athens, GA. March, 1997.
Building a model to test the capacity-speed hypotheses. (co-author) Paper presented at the meeting of the Eighth Annual Student Convention in Gerontology and Geriatrics, Athens, GA. April, 1997.
Harris, R., Iavecchia, H., Ross, L. and Shaffer, S. (1987). Microcomputer Human Operator Simulator (HOS-IV). Human Factors Society, Annual Meeting, 31st, New York, NY, USA. 19-23 Oct. 1987. pp. 1179-1183. 1987