A Cognitive Approach to Identifying Measurable Milestones for Programming Skill Acquisition

Jerry Mead
Computer Science Department
Bucknell University
Lewisburg, PA USA
mead@bucknell.edu

Simon Gray
Department of Mathematics
and Computer Science
College of Wooster
Wooster, OH USA
sgray@wooster.edu

John Hamer
Department of Computer Science
University of Auckland
Auckland, New Zealand
j.hamer@cs.auckland.ac.nz

Richard James
Department of Mathematics
and Computer Science
Rollins College
Winter Park, FL USA
rjames@rollins.edu

Juha Sorva
Computer Science Department
Helsinki University of Technology
Helsinki, Finland
jsorva@cs.hut.fi

Caroline St. Clair
Computer Science Department
North Central College
Naperville, IL USA
cstclair@noctrl.edu

Lynda Thomas
Computer Science Department
University of Wales
Aberystwyth, Wales
ltt@aber.ac.uk

ABSTRACT

Traditional approaches to programming education, as exemplified by the typical CS1/CS2 course sequence, have not taken advantage of the long record of psychological and experimental studies on the development of programming skills. These studies indicate a need for a new curricular strategy for developing programming skills and indicate that a cognitive approach would be a promising starting point. This paper first reviews the literature on studies of programming skills, cognition and learning, then within that context reports on a new formal structure, called an *anchor graph*, that supports curricular design and facilitates the setting of measurable milestones.

Categories and Subject Descriptors

K.3.2 [Computers and Education]: Computer and Information Science Education—*Computer science education, Information systems education, Literacy*

General Terms

Design, Languages

ITiCSE '06, June 26-28, 2006, Bologna, Italy.

Keywords

CS1, Programming, Cognitive Approaches, Curricular Planning

1. INTRODUCTION

Teaching programming has three basic components: curriculum, pedagogy, and assessment. A considerable body of literature shows that since the early days of programming education there has been concern on the part of educators about the programming abilities of their students. During this same period cognitive scientists, learning theorists, and computer scientists have studied the problems associated with learning to program. Unfortunately, their findings have not made their way into textbooks or curricular structures.

A related problem is that, while we expect that our students will learn to program, several studies tell us that we have no clear notion as to what level of programming skill acquisition our students should attain. Again, researchers have studied the problem of skill acquisition and of how to assess student performance but the research findings have not been applied widely.

In this paper, rather than focusing on new teaching techniques, we focus on a new strategy for curricular planning. The new strategy will facilitate the development of curricula within which existing and new teaching techniques can be more effectively applied and more easily assessed.

The new curricular planning structure we present is embodied in two new ideas: *anchor concept* and *anchor graph*, where an anchor graph is a graph with anchor concepts as nodes. An anchor graph brings together the idea of anchor concept and cognitive load to provide a structure within which course layout can be planned, multi-year goals can be set, and appropriate assessment

planned. In sum, anchor graphs are a cognitive approach to identifying measurable milestones for programming skill acquisition.

Section 2 reviews the history and background of the long-standing problems associated with programming education and also of the research into possible solutions. Section 3 reviews work carried out on curricular planning in computer science, in particular on how concept categorization has been done. Section 4 provides an overview of work in cognitive science, learning theory, and computer science on how the learning process occurs. In Section 5 we define the notions of anchor concept and anchor graph, provide an example graph for algorithmics, and discuss our ideas in the light of the cognitive and learning theories reviewed in Section 4. In Section 6 we discuss how the anchor graph idea can affect curricular planning and pedagogy. Section 7 presents possible extensions to the reported work.

2. BACKGROUND AND HISTORY

The record of experimental studies on the development of programming skills reveals that, despite the best intentions and efforts of the computer science education community, our programming students may or may not have the programming mastery expected of them. While the main problem has been studied extensively, a secondary problem has received little or no direct attention: computer science educators don't have a formal mechanism for defining what level of mastery they should expect of their students, after their first year or when they graduate.

2.1 The Student Performance Problem

Early work on student performance focused on identifying the particular problems that students have when learning to program. du Boulay's research [14] identified many of the areas that cause students trouble (e.g., the underlying *notional machine*, notation and algorithmic structures) as well as the common mistakes made in these areas (e.g., misapplication of analogy and over-generalization). Studies by Spohrer and Soloway [39, 40, 41] and also by Winslow [45] conclude that combining algorithmic structures into programs is problematic for most students.

The fact that these problems persist today is documented in more recent studies. The ITiCSE 2001 McCracken working group [25] investigated first-year student programming proficiency. Attempting to refine this work, the ITiCSE 2004 Lister working group [20] focused on students' capabilities to trace programs (see also Lister, et al. [22]). In another multinational study presented at SIGCSE 2006, Eckerdal et al. [16] analyzed the design work of graduating computer science students. These studies found that students could not program, trace programs, or design programs at acceptable levels.

Focusing on the "Why?," Buck and Stucki [8] identified as problematic the "too much too soon issue," created when educators overwhelm beginning students. Navrat [32] described the abstractness of our discipline as another factor contributing to students' difficulties in learning to program.

In short, over the past 25 years national and international studies have provided empirical indicators showing that learning to program is indeed challenging for most students. Thus, the problem is both long-standing and international in scale.

2.2 Assessing Programming Ability: Cognitive Approaches

While the authors have found no specific reference to formal techniques for goal setting within programming curricula, there has been work that can be applied to this problem. Dreyfus and Dreyfus [12], for example, describe a sequence of five stages of skill acquisition along a continuum from novice to expert. Their approach is completely general and favors no particular type of skills. When applied to programming, as Winslow [45] does, each stage characterizes a set of programming and problem solving skills we can expect a practitioner at that level to demonstrate. Studies of programmers and programming activities [36, 45] reinforce this view of gradual development.

Focusing on the cognitive process of skill acquisition, works by Cooper [10] and by Paas et al. [34] focus on the issue of the cognitive load of programming on novices and the impact of this load on learning. Mayer [24] addressed the issues of learning programming from a perspective based on Ausubel's work [2], concluding that transferring knowledge to new situations is critical.

On the assessment side, Biggs and Collis [5] define the Structure of the Observed Learning Outcome (SOLO) taxonomy. The SOLO taxonomy describes a hierarchy of complexity and sophistication that appears in student solutions and can be used to measure the level of mastery of a topic. Bloom [6] also produced an assessment taxonomy for identifying depth of learning. Numerous studies have described how these taxonomies can be applied to computer science pedagogy [21, 38].

To date these two avenues of research into how students learn and how they learn to program have not been effectively combined to produce more successful programming education strategies. Although several papers [4, 19, 30, 42] have identified pedagogical techniques that seek to improve aspects of programming instruction, a coherent approach recognizing the needs of novices and the cognitive demands of the discipline is still missing. On the promising side, the application of the Dreyfus stages to computer science [45] and the SOLO [5] and Bloom [6] taxonomies provides an attractive mechanism for identifying measurable goals or milestones for programming curricula.

2.3 The Way Forward

Recognizing the problems of programming education and based on the processes of learning and cognition, we present a formally-based approach to curriculum development that will facilitate setting measurable milestones and assessing those milestones. This approach hinges on two new ideas: the notion of an anchor concept (AC), a derivative of threshold concept introduced by Meyer and Land [27, 28] and further elaborated by Eckerdal et al. [15], and the notion of an anchor concept graph, whose structure is derived from ideas from cognitive load theory. Understanding of anchor concepts leads to mastery of the skills necessary to progress through the Dreyfus levels. An anchor concept graph ties together a collection of anchor concepts to minimize the mental demands on students while they are learning each concept.

3. CONCEPT CATEGORIZATION

Our interest in concept categorization derives from our desire to describe a general and formal strategy for defining curricula as "ordered" sets of concepts. While the authors have found no evidence in the literature for such a formal approach, there have been a few attempts to categorize concepts within computer science. From the world of computer science education research these include the 'fundamental ideas' of Schwill [37], and 'threshold concepts' as defined by Meyer and Land [28] and elaborated within the computer science context by Eckerdal et al.

[15]. Another perspective is provided by the IEEE/ACM Joint Task Force on Computing Curricula [33].

3.1 Schwill

Building on the ideas of Bruner [7], Schwill [37] argues that we should organise the CS Curriculum around certain 'fundamental ideas' that relate to fundamental notions, principles and ways of thinking that occur throughout the computer science curriculum. A fundamental idea satisfies the following four criteria:

Horizontal: can be seen in multiple ways and multiple areas within a discipline and can integrate multiple phenomena,

Vertical: can be taught at multiple levels of understanding, from primary school to beyond graduate school,

Sense: can be found in the actions and things of everyday life,

Time: have withstood the test of time, appearing throughout the domain's history.

The vertical and sense criteria make plain that fundamental ideas are quite general. Schwill proposes concepts such as algorithmization, language, and structured dissection as candidates for fundamental ideas.

Schwill does not suggest a way to organize fundamental ideas within a curriculum, but he does suggest that fundamental ideas be presented using Bruner's spiral principle [7]. That is, because fundamental ideas occur throughout a curriculum (indeed, according to the sense and time criteria, throughout one's life experiences), a student will see them several times, often within new contexts. Each exposure to a fundamental idea should strengthen the student's understanding of it and of all associated ideas.

3.2 Threshold Concepts

Meyer and Land [27, 28] introduced the notion of threshold concept as a particular basis for differentiating between core learning outcomes that represent seeing things in a new way and those that do not. As such, threshold concepts are conceptual gateways or portals that lead to a previously inaccessible way of thinking about something. Threshold concepts have the following characteristics:

Transformative: they occasion a significant shift in a student's perception about ideas within a discipline,

Irreversible: once learned, a student is unlikely to "unlearn" them,

Integrative: they expose the previously hidden interrelatedness of related concepts in ways that are new for the student,[1]

Troublesome: they are the places where students are likely to have trouble and become "stuck" because the concept may be conceptually difficult, counter-intuitive, involve language that is "alien," etc.,

Boundary markers: mastery of a threshold concept represents movement into a new conceptual area that was previously inaccessible.

Eckerdal et al. [15] have been examining threshold concepts in the context of computer science education and are currently performing a phenomenographic study of senior students to determine some candidate concepts. Their work is in its early stages, but the indications are that, while threshold concepts might have an interesting place in computer science curricula, they are not easy to identify. Furthermore, their rarity makes them less useful from a curricular point of view: there seems to be a lot of "learning space" to be accounted for between threshold concepts.

It is important to look at this idea in the context of curricular planning. When we examine closely the five characteristics of threshold concepts we find that they are not all relevant to curricular planning. The "transformative" and "integrative" characteristics seem relevant since they imply a connection to earlier and possibly following concepts, the implied sequence of concepts might be part of a curriculum. But the other three characteristics, "irreversible," "troublesome," and "boundary marker," seem to be more relevant to teaching.

3.3 ACM Curricula

The computer science education community has generated suggested curricula since 1968 with the most recent release in 2001 [33]. The latest set of guidelines identifies 132 individual thematic modules (called units) that encompass the computer science body of knowledge. The units are grouped into specific subfields (operating systems, programming fundamentals, etc.). Each unit is further described by a set of specific topics as well as a set of learning objectives to support assessment.

The guidelines provide six curricular models for introductory courses (such as imperative first, objects first, functional first) reflecting the broad set of opinions on how these courses should be approached.

The report does not recommend a single model as no ideal has yet been found. However, its authors believe it is possible to develop an introductory experience that meets the goals of introducing students to a set of fundamental concepts, facilitating development of cognitive models for these concepts, and encouraging students to develop skills necessary to apply their conceptual knowledge. The report identifies thirteen concepts to be covered in the introductory courses.

While this incredible volume of work provides an encapsulation of the current body of knowledge for the undergraduate computer science curriculum and recognizes the criticality of the introductory courses, the issue of cognitive difficulties with learning computer science concepts is not clearly addressed. The identified broad knowledge space needs a better guide for curricular development.

3.4 The Way Forward

In each of these curricular approaches a set of concepts is identified. The fundamental ideas of Schwill and the threshold concepts of Meyer and Land seem to occupy the two extremes for concept identification: fundamental ideas are quite general, identifying large subareas appearing in multiple places throughout a discipline's knowledge space, while threshold concepts are

[1] It may be that including the "integrative" characteristic is unnecessary, as it seems to be an artifact of the "transformative" characteristic. If a concept is "transformative" and integrates other concepts, then "integrative" would seem to be unnecessary; if on the other hand a concept is "transformative" and doesn't integrate other concepts then again "integrative" seems unnecessary. If a concept is "integrative" then it changes the student's view of the discipline, even if in only one corner — i.e., it is transformative. One can also doubt the necessity of the "irreversible" characteristic: if, as implied by the transformative characteristic, a student's view of the discipline is truly transformed, then how could that view be reversed?

much less common and rather specific, identifying concepts that hold key and unique places within a discipline's knowledge space. The ACM approach is decompositional (top-down) and fits somewhere in the middle. The concept identification strategy that we describe in this paper is formally based and derives from ideas from cognitive science.

4. COGNITION AND LEARNING

Our goal is to describe a curricular planning strategy which is formally based and takes advantage of recent work in cognitive science and learning theory. The purpose of this section is to provide background from these areas that will underpin our formal strategy. The section reviews work on anchoring ideas, schema theory, cognitive load theory, and constructivist theory.

4.1 Assimilation Theory and Anchoring Ideas

Anchoring ideas are a central concept in the assimilation theory introduced by Ausubel [2]. They are "specific, relevant ideas in a learner's cognitive structure that provide the entry points for new information to be connected" [13]. An anchoring idea is a cognitive construct that serves as an anchor to which new ideas can connect as new information is encountered by the learner. Ausubel sees knowledge of a domain as a dynamic hierarchy of anchoring ideas. Because the formation of these ideas is dynamic, the processes work differently for different people. This means that anchoring ideas are personal and may or may not be meaningful or "correct." As new data is gathered, four kinds of cognitive process modify these hierarchies [13]:

Derivative subsumption: a new example is added to an existing idea, adding new information to remember, but not new structure in the hierarchy of anchoring ideas.

Correlative subsumption: a new feature is associated with an existing anchoring idea, modifying the anchoring idea in the process.

Superordinate learning: a new, more abstract anchoring idea is synthesized from a number of existing anchoring ideas.

Combinatorial learning: a new anchoring idea is generated by analogy from an existing anchoring idea.

Ausubel's work on anchoring ideas provides a convenient structure for thinking about knowledge formation, but it is a structure within which the learner is relatively passive. The information is taken in and new ideas are formed or modified in a hierarchical way. How anchoring ideas are used by the learner does not seem to be addressed. Schema theory, on the other hand, provides a more realistic (non-hierarchical) mechanism for knowledge formation and describes how the learner uses schema for mental activities like recognition and problem solving [44].

4.2 Schema Theory

Schema theory was developed in reaction to Ausubel's assimilation theory and was meant "to bring precision to Ausubel's ideas" [13]. A schema is a cognitive construct that helps people store, organize, and understand information. It integrates the prototypical features of a number of individual experiences, becoming an abstraction of those experiences. Thus, rather than forming a hierarchical structure of data, schema seem to store meaning and relationships [44]. Since a schema is an abstraction of a collection of experiences, a learner can apply them in novel, but related contexts. Schemas are personal; a shared experience may lead to the encoding of different schemas for different people. This could be a source for misconceptions, as different learners blend shared experiences with existing schema differently.

Driscoll lists three ways in which new experiences affect schema formation [13], each of which corresponds more or less to one of the mechanisms for modifying an anchoring idea hierarchy.

Accretion: A new experience fits in well within an existing schema and while it is remembered as another instance consistent with prior experience, the schema is not significantly altered.

This corresponds to derivative subsumption.

Tuning: A new experience cannot be fully understood in the context of an existing schema, causing the schema to evolve to accommodate the new experience.

This corresponds to correlative subsumption.

Restructuring: A new experience is different enough that tuning an existing schema is not viable. A new schema is created.

This corresponds to combinatorial learning or superordinate learning, depending on the situation.

Schema are central to all our mental actions. Some schema require conscious attention to be activated — for example if one is asked to multiply two large numbers. Since most of us can't multiply large numbers immediately in our heads, they must be multiplied through our conscious control. Other schema are used automatically, meaning they are triggered without first being brought to working memory. When we walk, talk, read, or rattle off our phone number, the action is produced by the automatic triggering of schemas. Obviously, when schemas become automatic, their use will decrease the demands on working memory [43].

4.2.1 Schemas and Expertise

Schema formation is a natural part of learning and is key to the development of expertise and problem-solving ability. A key difference separating the novice from the expert in a domain of knowledge is that the expert has schemas that facilitate categorizing and processing information. When faced with new information, the expert can draw on these schemas to recognize similarities with prior experience and select suitable problem-solving strategies. This is facilitated by the fact, as described above, that schema organize meaning. In contrast, novices lack sufficient useful schemas for problem-solving and often have only disassociated bits of knowledge; the novice commonly resorts to trial-and-error or means-ends analysis as problem-solving methods [9, 18].

Since schemas are crucial in making an expert out of a novice, it is an important goal for instructors to facilitate their construction. Muller [30] suggests that in order to help students build schemas in a domain "one must first identify the major ideas in the field and then construct a curriculum that emphasizes learning to recognize these ideas, and which develops the skills and procedures for using them." Some current work in the field of computer science education, such as design patterns, roles of variables [19] and algorithmic patterns [30], can be viewed as attempts to support schema formation.

4.3 Cognitive Load Theory

Unlike assimilation theory and schema theory, which make no reference to how memory actually works, cognitive load theory is

based on a specific model of memory and its operation. In this model there are three forms of memory [10]:

Sensory: stores visual and auditory data. Visual data is retained for up to 0.5 seconds and auditory data for up to 3 seconds.

Working: receives data from long-term memory or from sensory memory and, based on conscious processing of the working memory contents, encodes the data in a form that can be stored in long-term memory. Working memory has a very small capacity which has been shown to be between 4 and 9, depending on circumstance.

Long-term: is an effectively unbounded storage space for encoded data. Encodings must be produced in working memory before being stored in long-term memory.

We will retain the terminology from the previous section and refer to the encoded chunks of information as schema.

Cognitive load is a structure representing the load imposed on a learner's working memory, that is the level of 'mental energy' or working memory capacity required to process a given amount of information [10].

Cognitive load can be created by characteristics of the learner, the given learning task, and the interaction between learner and task. From the perspective of memory, cognitive load is the sum of the following three kinds of working memory load [35]:

Intrinsic: the intrinsic difficulty of the material being learned. Material with high element interactivity leads to high intrinsic cognitive load as the elements cannot be learned without simultaneously considering other elements.

Germane: the amount of working memory resources the learner must devote to other data needed for the formation and internalization of a new schema.

Extraneous: the amount of working memory needed by instructional procedures and materials that are not necessary for a learning task.

For meaningful learning[2] to occur and schema to be formed and stored in long-term memory, new information must be processed by working memory and comprehended. When working memory is overburdened by a high cognitive load, the ability to process information is reduced and learning will be less effective.

4.3.1 Cognitive Load and Expertise

Winslow [45] distinguishes between "tasks" and "problems." A task has a goal with a known, or quickly identified, solution, while a problem is a goal that has no familiar solution. A novice solves problems slowly and clumsily, while an expert, in possession of useful schemas, does so relatively quickly and effortlessly. That is, what is a problem to a novice is just a task for an expert [45]. Cognitive load theory goes some way in explaining why schema are so important for fluent problem-solving.

A schema combines a number of elements and their interrelationships, and once internalized by a learner, can be brought to working memory and processed as a single element, thereby greatly reducing cognitive load [35]. The most useful schemas are those that are triggered automatically, without having to be brought into working memory. Without such high-level schema, a novice is forced to bring multiple lower-level schema into working memory and consciously apply them.

4.3.2 Cognitive Load and Pedagogy

From a pedagogical point of view, it is important to keep cognitive load low enough that it won't interfere with learning and the formation of schemas [10]. Intrinsic cognitive load cannot be affected, as it is an immutable characteristic of the material being taught. The instructor can, however, affect total load levels by adjusting germane and extraneous cognitive load [35]. While germane cognitive load is useful for the learner, extraneous cognitive load should be kept as low as possible. This can be accomplished by modifying teaching materials and examples [10], by using scaffolding, and by carefully sequencing and timing the introduction of new concepts. Muller [30] with her pattern-oriented instruction is also trying to decrease cognitive load.

It is important to locate such material within a curriculum that has high element interconnectivity and therefore high intrinsic cognitive load. Such material requires particular attention to ensure that total cognitive load is bounded to promote effective learning.

4.4 Constructivist Theory

The constructivist theory of learning claims that knowledge is actively constructed, not passively absorbed from textbooks and lectures [3]. It holds that a student's ability to understand new material depends on their existing knowledge, and hence that two students may come to construct different understandings from the same learning experience. The two conflicting views may not be equally viable, and learners must be challenged to test and justify their understandings and those of others. Shared meanings are thus arrived at through a process of social negotiation rather than individual study.

In language that by now sounds familiar, Piaget [13] talks about constructing new knowledge through assimilation and accommodation.

Assimilation: alignment of a new experience with one's existing knowledge framework of what is already "known."

Accommodation: occurs when a new experience is so different it cannot be aligned with existing knowledge, resulting in a restructuring of a portion of the framework.

With respect to learning, the role of prior knowledge is essential in the construction process. Furthermore, constructivist theory argues that students enter a course with a certain body of knowledge, and effective instruction should take that into account. Thus the role of the instructor is to facilitate the construction process, and not simply to "transfer knowledge" into the students.

4.5 The Way Forward

The connections among assimilation theory, schema theory, cognitive load theory and constructivist theory have been made clear in this section. It is important for our purposes, however, to point out a connection that may not be so clear between these theories and the notion of threshold concept. The mechanisms these theories describe for the construction of new knowledge have a strong relationship with the integrative and transformative characteristics of a threshold concept. Using the language of schema theory, integration occurs through accretion, tuning, or restructuring, depending on the complexity of the integration; transformation, on the other hand, occurs only through

[2] This use of the phrase "meaningful learning" is not meant to refer to Ausubel's notion of *meaningful learning* [2].

restructuring. It isn't clear what the connection to schema theory might be for the other three threshold concept characteristics. Thus the notion of threshold concept is at least partially supported by these cognitive theories. These connections are critical in the section that follows.

5. ANCHOR CONCEPTS

A curriculum can be viewed from two strongly related points of view: as a set of courses laid out in a partial pre-requisite order or as a partially ordered set of concepts to be taught. The first view is top-down and sees a course as an organizational mechanism that encapsulates the concepts to be addressed at particular times specified by the curriculum. The second view is bottom-up and allows the concept sequences to be determined by pedagogical considerations. While the top-down approach produces a layout of courses that can vary from one school to another, one would expect that the bottom-up approach would be more portable. Hence, the bottom-up approach is our strategy for curricular development.

Our work has been motivated by a desire to find a mechanism for the development of programming curricula that will result in improved student programming skills. To this end, we describe a way to identify sets of concepts and an ordering of the concepts based on minimum cognitive load. In this section we introduce a variation of threshold concept that we call *anchor concept* and describe how it can form the basis for curriculum design.

5.1 Defining Anchor Concept

The notion of threshold concept might seem an interesting approach to do concept selection, but, as implied earlier, agreeing on concepts satisfying all five characteristics will be difficult and there may not be enough threshold concepts on which to base a curriculum. There are two threshold concept characteristics, however, that seem to be more promising and relevant for curricular planning: integration and transformation.

We encapsulate these considerations in the following definition.

DEFINITION 1: (ANCHOR CONCEPT (AC))

An anchor concept *is a concept that is either* foundational,

> *i.e., it is a critical, basic concept in the knowledge domain, but not derivable in that domain*

or is both integrative

> *i.e., it ties together concepts from the knowledge domain in ways that were previously unknown*

and transformative

> *i.e., it involves a restructuring of schema, possibly integrating new information, resulting in the ability to apply what is known either differently and/or more broadly.*

The recursive nature of this definition is clear. A discipline will have certain foundational concepts, perhaps coming from other disciplines, that don't integrate previous concepts in the discipline and consequently can't be transformative. Foundational concepts must either be taught from scratch, drawing on concrete models or real-world situations, or brought in from other disciplines. An example from programming might be the concept of memory location or that of fetch-execute cycle (sequencing), which are not part of the programming discipline but are drawn from the structure of the von Neumann machine architecture. These two concepts can be taught by showing a model of a simple von Neumann machine. Another example would be the use of the concept "relation" from a discrete math course.

By this definition any threshold concept is also an anchor concept. So in expanding our attention to anchor concepts we lose nothing and get a pedagogically richer set of concepts from across a knowledge domain; additionally anchor concepts are easier to identify than threshold concepts.

5.2 Defining Anchor Graph

The notion of AC, however, does not stand alone. The fact that AC's are integrative encourages a natural ordering: if certain anchor concepts are integrated by another anchor concept, then the integrating concept should follow the others in order. This is exactly what we would expect to happen in the classroom. The student would first confront those concepts integrated by the AC, and then tackle the AC. Of course, having first learned the integrated concepts, the mental (cognitive) load of learning the AC should be lightened. Following this lead and borrowing from concepts of cognitive load theory, we define the following relation on anchor concepts.

DEFINITION 2 ($\not\geq$)

If a and b are anchor concepts then a $\not\geq$ *b if and only if a carries part but not all of the cognitive load in learning b.*

It is important to investigate the mathematical properties of this new relation. It should be clear that $\not\geq$ is irreflexive since it is not possible for a concept to carry cognitive load in learning itself! In addition, we can assume that $\not\geq$ is acyclic. If it had cycles then we could use the standard strategy to take the transitive closure of $\not\geq$ and use the resulting equivalence classes as our anchor concepts; the relation $\not\geq$ would be appropriately modified. The result is that it is our relation $\not\geq$ that determines a partial order on our set of anchor concepts and that relation determines a graph on the set of anchor concepts.

In a broader context, it is very important to understand that the foregoing development is completely general; like threshold concepts, the notion of AC is applicable to any area of study. In each different area the specific AC's will have to be carefully identified and linked via the $\not\geq$ relation. In a particular area it may be appropriate for AC graphs to be defined by a committee at a fairly high level, national or international, in order to promote confidence in the graphs produced.

It is also important to grasp the implications of the structure of an AC graph. The simplest way to state the implication is that from a cognitive load point of view, the cognitive load for an AC is *greater* than the sum of the cognitive loads carried by the subordinate concepts (those concepts with links to the AC under consideration), because in addition to the load required to understand the parts, the student must also spend mental energy to form the schema for the new concept.

To complete this section we present two examples, the first discussing the anchor concept *selection* from algorithmics and the second discussing an AC graph for algorithmics.

EXAMPLE 5.1 (Algorithmic Selection)

As an example, taken from a standard CS1 course, consider the algorithmic concept selection*. Consider the following concepts: boolean expression, machine-level branching and machine-level sequencing. Each of these concepts carries cognitive load for the concept of selection. In addition, it should be clear that selection, with its structured form, is more than simply the sum of the three supporting concepts; i.e., selection integrates the concepts of*

boolean expression, branching and sequencing; in addition, knowledge of selection transforms the student's view of programming by providing a structured strategy for attacking a large class of algorithmic problems. Since expression can be shown to be an anchor concept (in a similar manner) and branching and sequencing are foundational, we can conclude that selection is an anchor concept. We would also conclude that

boolean expression ⩾selection *and*

machine-level sequencing ⩾selection *and*

machine-level branching ⩾selection.

EXAMPLE 5.2 (AC Graph for Algorithmics)

Figure 1 shows an AC graph for algorithmics, where the foundational AC's are denoted with rectangles. The graph is meant to capture that component of programming education involved in algorithm design on the problem solving side.

The foundational concepts all represent concepts from machine organization. We found, in trying to build this graph, that without this base it was more difficult to determine where to start. We have concluded that in teaching imperative programming it is wise to start by teaching students the structure and operation of a very simple von Neumann machine. These foundational concepts provide an effective mental model with which to teach the more advanced concepts in the graph.

There is a bit of abstraction in this graph. The foundational concept "instruction" in fact represents all machine language instructions. Consequently, the link from "instruction" to "selection" is labeled "branch" because an understanding of the branching instructions will help understand the "selection" (and "repetition") concept.

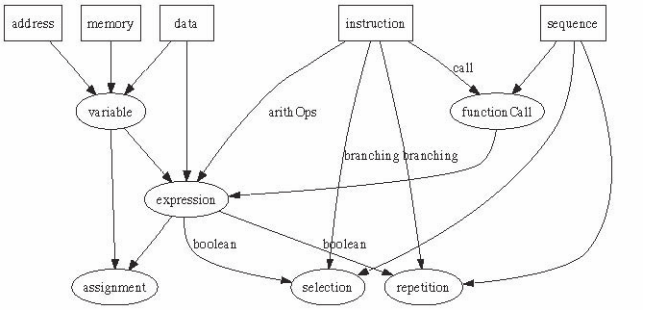

Figure 1: AC Graph for Algorithmics

Example 5.1 gave an idea of how one identifies and justifies a particular AC and how the links are made to it from concepts already learned. The other concepts in the graph are determined in a similar fashion. Clearly, for the algorithmic area the concept identification is quite intuitive. It should be noted that in each case, understanding a non-foundational concept will require more than simply understanding the concepts leading to it; it will require a certain amount of additional cognitive load to master the concept.

5.3 Anchor Concepts, Cognition, Learning

In this section we take a quick tour of the cognitive and learning theories discussed in Section 4 and show how they relate to our two new ideas of anchor concept and anchor graph.

5.3.1 Anchoring Ideas

Although Ausubel's notion of anchoring ideas has been supplanted by the more flexible notion of schema, it is instructive to look briefly at the difference between anchoring idea and our anchoring concept. For Ausubel, learning resulted in the construction of a cognitive structure as a hierarchical collection of anchoring ideas, and anchoring ideas were the ideas through which new information could be integrated into the structure. Thus anchoring ideas and assimilation theory describe a *dynamic* process of knowledge acquisition. Furthermore, since one's cognitive structure is the sum of one's own learning experiences, cognitive structures, and the anchoring ideas they contain, will differ from one person to the next. That is, they are *subjective*.

An anchor concept, on the other hand, is an *objective* and *static* piece of information from some knowledge domain, and an anchor graph is an *objective* and *static* representation of a sequencing of concept acquisition within the domain. The actual learning of the concepts will, of course, vary from individual to individual, but that is independent of the concepts themselves.

5.3.2 Schema Theory

Like anchoring ideas, schema are an *individualized* representation of knowledge and are *dynamically* created. But where anchoring ideas are linked in a hierarchical structure, schema are created and linked to reflect the (perceived) relationships between the schema and the new information learned. In this way schema theory more closely matches our notion of transformation, because the creation of a new schema is more than the integration (sum) of its parts. As Muller has suggested [30], we need to identify and order the major ideas in a knowledge domain to help students build effective and accurate schemas. AC's and AC graphs are ideally suited to this task.

5.3.3 Cognitive Load Theory and Schema Development

Cognitive load theory is concerned with the limitations of working memory and the construction of knowledge in light of those limitations. One difficulty facing educators is recognizing places where concept interconnectivity is high (thus has high cognitive load), making schema formation more problematic and error prone. This problem is compounded by the fact that as experts, educators have internalized schemas that conceal the complexity of the material to be learned. What may seem to an expert a simple learning task requiring few elements of interactivity, may in fact require schemas students have not yet (fully) developed. In addition, educators must be aware of the fact that pedagogical decisions affect extraneous cognitive load, potentially limiting the amount of short-term memory available for concept acquisition.

An anchor concept graph highlights the concept interactivity involved in learning an anchor concept by showing the number of links leading to the concept. An anchor concept graph directs the focus to the minimal set of necessary concepts students need to be aware of while learning a new goal concept. Any other concepts introduced by the educator not linked to the goal concept may increase extraneous cognitive load. Thus, an anchor concept graph is a tool an educator can use to develop a course that incorporates cognitive load theory. The teaching techniques advocated by many of the cognitive learning theories are supported by anchor concept graphs. The graphs show points of interest for schema formation and the mental "cost" of learning the concepts that are

part of the schema, thus identifying paths through the material that will be more effective for learning.

5.3.4 Constructivist Theory

Constructivist theory says that as learners experience (engage with) new information, they incrementally build cognitive structures to store that information. Significantly, this construction occurs within the context of what is already "known" by the learner. It is well known that students commonly construct idiosyncratic knowledge and alternative mental models of systems even when they are not explicitly taught a model [3, 14].

The difficulty for an educator is twofold. First, students enter a course with an established body of knowledge, some of which may be incorrect. Second, an educator must select which concepts learners should engage in and determine their order to promote effective construction of knowledge

Anchor concepts and their graphs can address both challenges. The foundational concepts for an AC graph represent what should be known to make progress on the AC's in the graph. Ensuring that students have a correct understanding of those foundational concepts goes some way toward addressing the "pre-existing knowledge problem." The AC's themselves and the AC graphs highlight the knowledge relevant to learning a collection of concepts within a domain, addressing the selection and ordering issue.

6. CURRICULAR ISSUES

In this section we discuss curricular issues and AC graphs: there are three important issues. First, AC graphs may be of theoretical interest, but without actual graphs they will have little impact on problems faced by a discipline. Figure 1 gives an AC graph for algorithmics. Here we will present two more graphs, focusing on aspects of object-orientation. Second, AC graphs provide a powerful tool for curriculum planning, especially

- specifying program structure,
- identifying course as well as program goals, and
- determining course and program assessment strategies.

We discuss each of these issues in turn. Finally, we will discuss the connection between our more theoretical graph production and pedagogy.

6.1 Establishing AC Graphs

An important characteristic of AC's and AC graphs is that they are completely general. They should be applicable to any discipline or disciplinary component. But ultimately our stated interest is in curricula for programming education. The working group worked on defining three AC graphs, one for algorithmics and the other for object oriented design.

During the process of producing the graphs we discovered strategies that seem to make the process simpler and more effective. The first is the importance of selecting the foundational concepts. It seems that in producing a curriculum for programming, the foundational concepts should come from the underlying principles on which the target language is based. So for an imperative-type language, the consensus of the authors was that the foundational concepts should reflect the von Neumann architecture — perhaps at a very simplistic level. For a functional language, perhaps the foundational concepts are simply function, function combination, and function evaluation, reflecting the structure of the λ-calculus or recursive function theory.

A second strategy we discovered is to try to isolate different aspects of an area and generate graphs for each of these areas. A complete graph might contain links from the graphs for the different areas. Again from programming, we found that concepts taught in a CS1 course, for example, come from different areas such as algorithmics, programming languages (e.g., scope, parameter), design (e.g., decomposition), and pragmatics (e.g., compiling).

A third problem we encountered was keeping a focus on curriculum rather than teaching. It was easy to let teaching strategies slip into the graph of concepts to be taught. One benefit of the exercise for all group members was to observe this dichotomy.

Finally, we found that determining a graph is a non-trivial activity. The process of isolating concepts and agreeing not only on the concepts, but also on how they are laid out in the graph is time-consuming and combative — agreement is not always easy to arrive at. Our conclusion is that AC graph determination should at least be reviewed by national or international professional boards.

6.1.1 AC Graph for Object Oriented Programming

We have commented on the difficulty in determining AC graphs. Here is an example illustrating that difficulty as well as the insight gained from the exercise.

While we found the algorithmics graph *relatively* straightforward, we found it much more difficult to determine where to start with the object oriented graph. After many false starts, we realized that we didn't have a good idea of what the foundational concepts should be. This was not altogether surprising, given the difficulties others have had with this problem [1].

We made some progress once we identified the problem solving side, not the programming language side, as our main interest. Consequently, we took as our foundational view the problem domain and from this view came the foundational concepts along the top of the AC graph in Figure 2. While the graph is incomplete, it is reasonable and seems headed in the right direction.

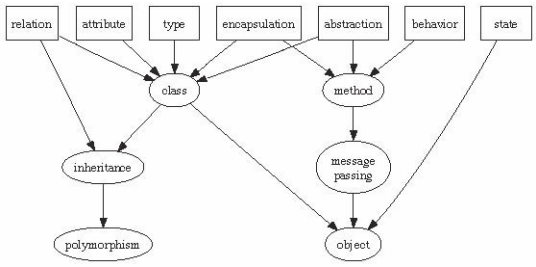

Figure 2: AC Graph for Object Orientation – Design Perspective

An alternative approach comes by taking a runtime view of object oriented programming, as we did with algorithmics. Unfortunately the runtime environment is much more complex than the von Neumann machine and might be difficult to present to students just starting out. The runtime environment typically has class tables, holding static attributes for the class, objects holding the dynamic state, a hierarchy of method tables, and a register "this" holding a reference to the object that most recently

received a message. A preliminary graph from the runtime point of view is presented in Figure 3.

The conclusion drawn from this example is that selecting the foundational view is critical. We learned from our attempt at the runtime view that this might not be the best place to start, whereas we found this dynamic view was quite natural for algorithmics.

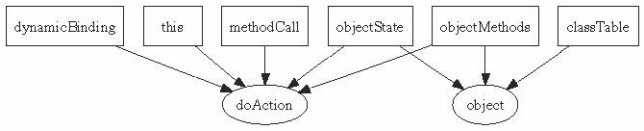

Figure 3: AC Graph for Object Orientation – Runtime Perspective

6.2 Anchor Graphs and Curricular Planning

The notions of anchor concept and AC graph were developed in part to provide a more structured environment in which to think about curricular issues. Once constructed, an AC graph can act as a structure for answering some important questions: what topics should different courses cover, what skill acquisition targets can reasonably be set for courses and programs, what assessment is appropriate for courses and programs?

6.2.1 Course design

The adoption of an AC graph by a program will facilitate determining the content of courses in the program. For the traditional required courses it should be a matter of identifying a subgraph with a kind of connectivity property: given a concept, each concept directly linked to it is either in the subgraph itself or in the subgraph of a required course. Notice that this strategy also helps define a prerequisite structure.

There is a pragmatic point to be made here. Having determined course boundaries, an instructor will know the concepts to be covered and how they relate to earlier courses and to courses that follow in the prerequisite chain. The AC graph (actually subgraph) for a particular course does not have to dictate the order in which topics are presented to students — a particular concept could be introduced earlier and then revisited later. However, the course graph will help the instructor understand the cognitive cost of introducing the concept and may help the instructor plan an appropriate introduction.

6.2.2 Course and program goals and assessment

While it may be possible for many schools to adopt the same AC graph and similar course structures, it is much less likely that these same schools will have, for a particular concept, the same expectations of student skill acquisition. The AC graph again provides a convenient structure. There are two issues here: how to specify skill acquisition goals and how to manage them for a course or program.

A standard taxonomy for skill acquisition is that introduced by Dreyfus and Dreyfus [12]. The taxonomy has five stages with each stage being characterized by specific capabilities.

Novice: A novice learns basic facts, terminology, and rules and how to apply them in well-defined circumstances. In this sense, everything novices do is largely "context free;" they are not asked to reason about the rules.

Advanced Beginner: The advanced beginner starts to develop a feel for the rules through repeated practical application. The elements become "situational;" the student is beginning to understand the use of concepts/rules in situations that are similar to those from prior examples. This represents a gradual extension of the student's learning from the controlled environment of the "context free" examples to novel, but familiar applications ("situations"). A goal of this stage is to develop more generalized understandings of the rules and when to apply them, leading naturally to the next stage.

Competence: Being competent means that one has a deep enough understanding of the rules to know when they are applicable and how to apply them in novel situations. It also means knowing how to create a plan; to look for and identify the facts relevant to a situation so that a decision can be made about which rule(s) to apply. Thus this level is characterized by conscious problem-solving.

Proficiency: The performer at this level has a more refined and internalized sense of the rules so is able to do a better job of filtering and evaluating the facts in a situation. A developing intuition (involving pattern matching of relevant facts against generalized experiences) leads quickly to suggestions for action, then conscious analysis is used to evaluate the alternatives.

Expert: Through extensive experience and reflection, experts produce increasingly abstract representations and become better at mapping novel situations to these internalized patterns. Experts often find it difficult to express their decision-making process in words.

When teaching a particular skill, the instructor can specify one of the five stages to indicate the expected capability of the student at the end of a module or course.

The AC graph structure provides a convenient setting for determining and recording skill acquisition expectations for a concept, course, or program. One approach to program goal setting could be as follows. Make one copy of the program's AC graph(s) for each year that a student progresses through the program. On the year one graph one would annotate each concept with a Dreyfus stage (or mark it as not covered) expected of students for that concept. Each concept in the graph would have an annotation. For the second year graph one would repeat the process, again for every concept in the graph. This means that some concepts might have the same stage annotation for both years, but the second year's annotation could be different. This same process is repeated for each of the remaining years of the program (one or two more years). On examining the three or four graphs, one would gain insight into the department's impression of the difficulty of various concepts and the pace of concept acquisition.

Since program and course goals can be effectively specified with an AC graph, it will not come as a surprise that assessment can be organized in the same way. By understanding the expected Dreyfus stage for a particular concept, the instructor of a program can decide on appropriate assessment strategies based on commonly used and understood assessment taxonomies such as Bloom [6] and SOLO [5].

Educators often speak about the desire that their students learn a particular concept during a course. The use of the Dreyfus stages and AC graph is a way to give precision to the otherwise very fuzzy notion "to learn." In addition, the assessment is a mechanism to determine if the expectation is being met or if the expectation is even reasonable.

6.3 Anchor Graphs and Pedagogy

Besides providing a convenient context for curricular planning, AC graphs also provide support for some modern teaching theories. In this section we discuss the connection between AC graphs and *scaffolding* and *pragmatics*.

6.3.1 Scaffolding

Scaffolding, introduced by Wood, Bruner and Ross [46], is a technique that uses a building block approach to learning and is currently advocated by educators within the field of computer science. Educators engage students in a collaborative manner by providing support structures for learning. These support structures ("scaffolds") help students complete tasks and develop knowledge structures, which students often have difficultly doing on their own. Many scaffolding techniques have been used in programming courses to help students understand difficult material [42]. Some of the requirements for using scaffolding techniques include providing clear directions, having a clear purpose, using methods that keep students on task, and providing intermittent assessment with feedback.

Anchor concepts and the use of an anchor concept graph provide the structure necessary to help an educator implement scaffolding techniques. An anchor concept graph is a structure that shows the required building blocks needed to understand a concept. Thus, the graph highlights locations of learning that may benefit from a scaffolding approach. If the purpose is for students to develop a knowledge structure represented by a node in the graph, which will be called the goal node, students must first traverse the graph developing knowledge structures for the intermediate nodes along the path to the goal node. With a well-defined anchor concept graph the direction is clear and the purpose is well-defined even if there are multiple paths to a goal node. Educators can also use this structure to keep students on track by making sure that paths that take students away from the goal node are not traversed. Intermediate nodes are also ideal locations for providing intermittent assessment, an important feature of scaffolding techniques.

6.3.2 Pragmatics

In a classic paper on the difficulties students must overcome in learning to program, du Boulay [14] points out that, while syntax and semantics are two important aspects of programming, they are by no means the only ones. He points out that issues such as mastering the *orientation* and *pragmatics* of programming are themselves important: what is programming for, what kinds of problems can it solve, what artifacts are produced and how are they tackled? For today's students these various aspects might include how to compile and run a program, how to use an IDE for program development, or how to test and maintain a program.

Eckerdal, Thuné and Bergland interviewed students in order to understand what learning to program meant to them [17]. The paper used the term *programming thinking* to refer to the issues raised by du Boulay; they report that students indicate considerable difficulties in acquiring this pragmatic knowledge.

Seeing pragmatics as an important area in teaching programming, the working group posited the following list as possible candidates for being anchor concepts.

- Program Structure
- Compilation
- Program Errors (compile and run-time)
- Software Lifecycle
- Problem solving techniques

In addition, an understanding of the behavior of the world in which we are developing the program is crucial. This may be what du Boulay calls the *notional machine* on which programs are run. This might lead to candidate fundamental anchor concepts such as sequence of execution and roles of variables [4, 19].

7. FUTURE WORK

The notions of anchor concept and anchor graph need to be further explored. In particular, a process is needed by which AC's can be identified and then formed into an AC graph. Since it is our expectation that, for a particular subject area, different sets of foundational concepts can give rise to different AC graphs, it will also be useful to have a process for identifying appropriate graphs. With these tools in hand, AC graphs can be constructed for other subject areas of computer science. In the same spirit in which threshold concepts have been investigated, it would be very interesting to apply the anchor concept idea to other disciplines.

As a next step, several research questions are proposed to gather data about anchor concepts and AC graphs:

1. Is there agreement among programming faculty on possible anchor concepts for CS1 and CS2?
2. Is there agreement on the organization of these concepts into AC graphs?
3. Variation theory says that a concept can be seen as having multiple dimensions and that these dimensions are key to student learning [11]. Motivated by the Meyer, Land and Davies investigation of connections between variation theory and threshold concepts [29], it would be interesting to explore connections between variation theory and AC's. For example, can we identify dimensions of an AC and, if so, do the dimensions of an AC give hints to a teacher for appropriate teaching strategies for the AC? Is variation theory an appropriate mechanism for dealing with pragmatics in programming?
4. If given a list of AC's, what AC graphs would students from CS1 and CS2 courses produce? What other concepts might students add to these graphs?
5. What changes can be seen in the AC graphs produced by students between the end of CS1 and CS2? Later courses? We would expect that as students progress through the curriculum, they would produce more complete graphs with fewer misconceptions, and that their graphs would be quite similar to those produced by programming faculty [31].
6. Do the results from independent assessment instruments correlate with the AC graphs drawn by students?

8. CONCLUSIONS

The goal of this working group was to address the following problem: What measurable milestones should be set within programming curricula to produce student programmers who meet our expectations? This report presents a formal structure, the *anchor graph*, which facilitates curricular planning and provides a context within which to set curricular goals and plan for their assessment. The anchor graph is based on another new idea, the *anchor concept*, which integrates and transforms earlier knowledge. The structure of an anchor graph is based on the idea that an anchor concept with a directed link to another anchor

concept carries cognitive load for learning the new concept. Three example anchor graphs have been created and presented in the report to illustrate both the potential of anchor graphs and the process of developing them.

9. POSTSCRIPT: TESTABLE, REUSABLE UNITS OF COGNITION

Shortly after the report of this working group was submitted for publication review, a member of our group came across Bertrand Meyer's article "Testable, Reusable Units of Cognition" [26] in which Meyer describes a system for organizing concepts that has several similarities to the notions of anchor concept and anchor concept graph developed by the working group and described in this report.

In his article Meyer describes a truc (Testable, Reusable Unit of Cognition) as "an entity that embodies a collection of concepts, operational skills, and assessment criteria" (p. 20). As such, trucs form a catalog of atomic elements found within the topics that make up an educational program. Trucs are described using a design pattern template consisting of: Name, Alternative names, Dependencies, Summary, Role, Applicability, Benefits, Examples, Common confusions, Pitfalls, and Tests of understanding. Trucs that are part of some field of knowledge can be grouped into clusters and related through dependencies, where concept A *depends on* concept B if understanding A requires previously having mastered B.

The similarities between anchor concepts and trucs, AC graphs and dependency graphs are obvious. We note here a few differences we believe are important, recognizing that only experience developing and using these "concept graphs" will tell us whether the differences are significant. More importantly, perhaps, such experience may suggest how these two approaches might best be blended.

The first clear difference is that we include the notion of foundational concepts where the foundational concepts are those that must be understood before tackling the anchor concepts in the AC graph. This notion is missing, but perhaps implicit, in Meyer's dependency graph. Relatedly, we believe that an entire AC graph may be treated as foundational for some other AC graph and that this is an important relationship to make explicit.

The second difference is the way in which concepts are related. Meyer's notion hinges on the idea that one concept 'depends' on another in the sense that understanding one concept can depend on understanding other concepts. This notion of truc and dependency are presented as intuitive; no formal definitions are presented. Anchor concepts, on the other hand, are derived from the notion of threshold concept with the well-established cognitive load theory being critical in the definition of the relation '≫that relates anchor concepts.

The third difference is in the structure of a graph. Meyer stresses that dependency graphs must not have transitive subgraphs, but gives no theoretical basis for this requirement. There is an intuitive appeal to this restriction and indeed it is one that the working group considered, but there are good reasons to allow this kind of relation. For example, in the AC graph of Figure 1 there is a transitive subgraph involving the AC's *variable*, *expression*, and *assignment*. This structure is acceptable within the AC graph context because different dimensions of the concept *variable* carry cognitive load for *expression* (r-value) than for *assignment* (l-value) [23]. This allows for a more nuanced understanding of the concepts involved.

Finally, we can point to a general difference in approach. As one reviewer of this report points out, Meyer appears to have developed trucs intuitively. The structure and content of this working group report accurately reflects the intellectual journey the working group took as it sought to understand the issues underlying learning to program (or, indeed, learning in *any* area). The notions of anchor concept and anchor concept graph grew out of our readings and discussions and are grounded in theories of cognition and learning.

10. REFERENCES

[1] Deborah J. Armstrong. The quarks of object-oriented development. *Communications of the ACM*, 49(2):123–128, 2006.

[2] David Ausubel. *Educational psychology; a cognitive view.* Holt, Rinehart and Winston, New York, New York, 1968.

[3] Mordechai Ben-Ari. Constructivism in computer science education. *Computers in Mathematics and Science Teaching*, [in press].

[4] Mordechai Ben-Ari and Jorma Sajaniemi. Roles of variables as seen by CS educators. In *ITiCSE'04: Proceedings of the 9th Annual SIGCSE Conference on Innovation and Technology in Computer Science Education,* pages 52–56, New York, NY, USA, 2004. ACM Press.

[5] J. B. Biggs and K. F. Collis. *Evaluating the Quality of Learning: The SOLO Taxonomy.* Academic Press, New York, New York, 1982.

[6] B.S. Bloom. *Taxonomy of Educational Objectives, Handbook I: The Cognitive Domain.* David McKay Co Inc., New York, New York, 1956.

[7] Jerome Bruner. *The Process of Education.* Harvard University Press, Cambridge MA, 1960.

[8] Duane Buck and David J. Stucki. Design early considered harmful: graduated exposure to complexity and structure based on levels of cognitive development. In *SIGCSE'00: Proceedings of the 31st SIGCSE Technical Symposium on Computer Science Education,* pages 75–79, New York, NY, USA, 2000. ACM Press.

[9] M. Chi, R. Glaser, and E. Rees. Expertise in problem solving. In R. Sternberg, editor, *Advances in the Psychology of Human Intelligence,* pages 7–75, Hillsdale, N. J., 1982. Erlbaum.

[10] Graham Cooper. Research into cognitive load theory and instructional design at unsw. http://education.arts.unsw.edu.au/CLT_NET_Aug_97.html, December 1998.

[11] Z. P. Dienes. The growth of methematical concepts in children through experience. *Educational Research*, 2(1), 1959.

[12] H. Dreyfus and S. Dreyfus. *Mind Over Machine: The Power of Human Intuition and Expertise in the Era of the Computer.* Free Press, New York, NY, USA, 1986.

[13] Marcy P. Driscoll. *Psychology of Learning for Instruction.* Allyn and Bacon, Needham Heights, MA, 2nd edition, 2000.

[14] Benedict du Boulay. Some difficulties of learning to program. *Journal of Educational Computing Research*, 2(1), 1986.

[15] Anna Eckerdal, Robert McCartney, Jan Erik Moström, Mark Ratcliffe, Kate Sanders, and Carol Zander. Putting threshold concepts into context in computer science education. In *ITiCSE'06: Proceedings of the 11th Annual SIGCSE Conference on Innovation and Technology in Computer Science Education,* pages 103–107, New York, NY, USA, 2006. ACM Press.

[16] Anna Eckerdal, Robert McCartney, Jan Erik Moström, Mark Ratcliffe, and Carol Zander. Can graduating students design software systems? In *SIGCSE'06: Proceedings of the 37th SIGCSE Technical Symposium on Computer Science Education,* pages 403–407, New York, NY, USA, 2006. ACM Press.

[17] Anna Eckerdal, Michael Thuné, and Anders Berglund. What does it take to learn 'programming thinking'? In *ICER'05: Proceedings of the 2005 International Workshop on Computing Education Research,* pages 135–142, New York, NY, USA, 2005. ACM Press.

[18] Kate Ehrlich and Elliot Soloway. An empirical investigation of the tacit plan knowledge in programming. In J. Thomas and M. L. Schneider, editors, *Human Factors in Computer Systems,* pages 113–134, Norwood, N.J., USA, 1984. Erlbaum.

[19] Marja Kuittinen and Jorma Sajaniemi. Teaching roles of variables in elementary programming courses. In *ITiCSE'04: Proceedings of the 9th Annual SIGCSE Conference on Innovation and Technology in Computer Science Education,* pages 57–61, New York, NY, USA, 2004. ACM Press.

[20] Raymond Lister, Elizabeth S. Adams, Sue Fitzgerald, William Fone, John Hamer, Morten Lindholm, Robert McCartney, Jan Erik Moström, Kate Sanders, Otto Seppälä, Beth Simon, and Lynda Thomas. A multi-national study of reading and tracing skills in novice programmers. In *ITiCSE-WGR'04: Working group reports from ITiCSE on Innovation and Technology in Computer Science Education,* pages 119–150, New York, NY, USA, 2004. ACM Press.

[21] Raymond Lister and John Leaney. Introductory programming, criterion-referencing, and Bloom. In *SIGCSE'03: Proceedings of the 34th SIGCSE Technical Symposium on Computer Science Education,* pages 143–147, New York, NY, USA, 2003. ACM Press.

[22] Raymond Lister, Beth Simon, Errol Thompson, Jacqueline Whalley, and Christine Prasad. Not seeing the forest for the trees: Novice programmers and the SOLO taxonomy. In *ITiCSE'06: Proceeding of the 11th Annual SIGCSE Conference on Innovation and Technology in Computer Science Education,* pages 118–122, New York, NY, USA, 2006. ACM Press.

[23] Kenneth Louden. *Programming Languages: Principles and Practice.* Course Technology, second edition, 2002.

[24] Richard E. Mayer. The psychology of how novices learn computer programming. *ACM Computing Surveys,* 13(1):121–141, 1981.

[25] Michael McCracken, Vicki Almstrum, Danny Diaz, Mark Guzdial, Dianne Hagan, Yifat Ben-David Kolikant, Cary Laxer, Lynda Thomas, Ian Utting, and Tadeusz Wilusz. A multi-national, multi-institutional study of assessment of programming skills of first-year CS students. In *ITiCSE-WGR'01: Working group reports from ITiCSE on Innovation and Technology in Computer Science Education,* pages 125–180, New York, NY, USA, 2001. ACM Press.

[26] Bertrand Meyer. Testable, reusable units of cognition. *IEEE Computer,* April 2006.

[27] Jan Meyer and Ray Land. Threshold concepts and troublesome knowledge (2): Epistemological considerations and a conceptual framework for teaching and learning. *Higher Education*, 49:373–388, 2005.

[28] Jan Meyer and Ray Land. Threshold concepts and troublesome knowledge: Linkages to ways of thinking and practising within the disciplines. *ETL Project: Occasional Report 4*, May 2003.

[29] Jan Meyer, Ray Land, and Peter Davies. Threshold concepts and troublesome knowledge (4): issues of variation and variability. *Threshold Concepts within the Disciplines Symposium*, August 2006.

[30] Orna Muller. Pattern oriented instruction and the enhancement of analogical reasoning. In *ICER'05: Proceedings of the 2005 International Workshop on Computing Education Research,* pages 57–67, New York, NY, USA, 2005. ACM Press.

[31] Jane Gradwohl Nash, Ralph J. Bravaco, and Shai Simonson. Assessing knowledge change in computer science. *Computer Science Education,* 16(1):37–51, 2006.

[32] Pavol Navrat. Hierarchies of programming concepts: Abstraction, generality, and beyond. *SIGCSE Bulletin*, 26(3), 1994.

[33] Joint Task Force on Computing Curricula. Computing curricula 2001 for computer science. http://acm.org/education/curricula.html, December 2001.

[34] Paas, Touvinen, Tabbers, and Van Gerven. Cognitive load measurement as a means of advancing cognitive load theory. *Educational Psychologist,* 38(1), 2003.

[35] Fred Paas, Alexander Renkl, and John Sweller. Cognitive load theory and instructional design: Recent developments [introduction to special issue]. *Educational Psychologist,* 38(1), 2003.

[36] Anthony Robins, Nathan Rountree, and Janet Rountree. Identifying the danger zones: Predictors of success and failure in a CS1 course. *Computer Science Department Technical Report OUCS-2001-07*, 2001.

[37] Andreas Schwill. Fundamental ideas in computer science (revised). *Zentralblatt für Didaktik der Mathematik*, 1, 1993.

[38] T. Scott. Bloom's taxonomy applied to testing in computer science. In *Proceedings of the 12 Annual CCSC Rocky Mountain Conference.* Consortium for Computing Sciences in Colleges, October, 2003.

[39] E. Soloway and James C. Spohrer. *Studying the Novice Programmer.* Lawrence Erlbaum Associates, Inc., Mahwah, NJ, USA, 1988.

[40] Elliot Soloway, Jeffrey Bonar, and Kate Ehrlich. Cognitive strategies and looping constructs: an empirical study. *Communications of the ACM,* 26(11):853–860, 1983.

[41] James C. Spohrer and Elliot Soloway. Novice mistakes: are the folk wisdoms correct? *Communications of the ACM,* 29(7):624–632, 1986.

[42] Lynda Thomas, Mark Ratcliffe, and Benjy Thomasson. Scaffolding with object diagrams in first year programming classes: some unexpected results. In *SIGCSE'04: Proceedings of the 35th SIGCSE Technical Symposium on Computer Science Education,* pages 250–254, New York, NY, USA, 2004. ACM Press.

[43] Juhani Touvinen. Optimising student cognitive load in computer education. In *Proceeding of the Australasian Conference on Computing Education,* volume 91(2), 2000.

[44] Sharon Alayne Widmayer. Schema theory: An introduction. http://chd.gse.gmu.edu/immersion/knowledgebase/strategies/cognitivism/SchemaTheory.htm.

[45] Leon E. Winslow. Programming pedagogy — a psychological overview. *SIGCSE Bulletin,* 28(3):17–22, 1996.

[46] D. Wood, J. Bruner, and G. Ross. The role of tutoring in problem solving. *Journal of Child Psychology and Psychiatry,* 17(2):89–100, 1976.