Suggestions for Graduated Exposure to Programming Concepts Using Fading Worked Examples

Simon Gray
College of Wooster
Wooster, OH
sgray@wooster.edu

Caroline St. Clair
North Central College
Naperville, IL
cstclair@noctrl.edu

Richard James
Rollins College
Winter Park, FL
rjames@rollins.edu

Jerry Mead
Bucknell University
Lewisburg, PA
mead@bucknell.edu

ABSTRACT

The fading worked example has emerged from cognitive load theory as an effective strategy for lowering cognitive load in the novice phase of skill acquisition and has been shown to be effective in mathematics, engineering and psychology. This theoretical paper proposes the application of the fading worked example strategy to programming education.

Categories and Subject Descriptors

K.3 [**Computers & Education**]: Computer & Information Science Education – *Computer Science Education*

General Terms

Algorithms, Design

Keywords

Cognitive Load Theory, Fading Worked Examples

1. INTRODUCTION

Instruction typically focuses on two types of knowledge: declarative (learning terms, facts, rules and principles) and procedural (the strategic application of declarative knowledge to the solution of a problem) [21]. Within the context of learning to program, declarative knowledge includes the structure of the language and pragmatic information such as use of program development tools. Procedural knowledge, then, is the application of declarative knowledge to develop software solutions and includes skills such as design, analysis, verification and debugging.

Clearly, like problem-solving in domains such as mathematics, engineering and medicine, computer programming (the combination of declarative and procedural programming knowledge) is an intellectual task requiring the development of a variety of cognitive skills over many years of practice [31]. The development and assessment of cognitive skills, including those necessary for effective programming, have been described in several reports over the years; for example, [9] and [31] on the developmental side and [5] and [4] on the assessment side. While the developmental phases and their characteristics differ in these reports, what they have in common is a 3-phase sequence: novice, intermediate, final (using the terminology in [31]). In computer science, the issues associated with teaching programming in these phases span several decades and are well-documented (a representative sample includes [32, 13, 25, 15, 27]). During this period, several research projects identified cognitive problems associated with skill acquisition, especially in the context of programming, but the results of these cognitive studies have not, for the most part, made their way into CS1 and CS2 courses [14, 30].

In 1988 John Sweller described a model of memory and used it to understand how the load on memory resources during problem solving impacts learning [28]. Since then his Cognitive Load Theory (CLT) has been used to understand the cognitive load of various teaching techniques at different points in the skill acquisition process [8, 11, 19, 29]. An interesting research direction has recently emerged deriving from the development of *fading worked examples* (FWE) as an effective strategy for lowering cognitive load in the novice and intermediate phases of skill acquisition.[1] FWEs derive from an important discovery of CLT that studying partially worked examples provides better learning results for novices than only working problems from scratch or studying completely worked examples [28, 22]. FWEs have been applied successfully in areas such as mathematics [8, 19], engineering [18], and psychology [23].

Our conjecture here is that part of the reason students don't acquire the desired programming skills is that they develop only near transfer (the ability to apply knowledge from the learned scenario to one similar in structure) due to cog-

[1]The meaning of "novice" is relative. One can have experience programming with common container types, but be a novice with respect to graph algorithms. By "novice" we mean here a student with no programming background entering a first programming course.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.
ICER'07, September 15–16, 2007, Atlanta, Georgia, USA.
Copyright 2007 ACM 978-1-59593-841-1/07/0009 ...$5.00.

nitive overload. We propose that by appropriately decomposing the programming process, it is possible to identify smaller components whose cognitive load can be controlled and that these components can be presented through FWEs.

Our identification of "smaller components" is based on two concept categories. First we have the basic constructs from which algorithms are constructed: selection, iteration, and subroutine call. Second we have critical dimensions of programming, including design, implementation and semantics. Each of the constructs can be paired with each dimension (e.g., <selection, design>, <iteration, semantics>, and so on) thus creating a set of "smaller components." Each of these components then becomes the focus for a set of FWEs. This approach, driven primarily from the algorithm construct level, is in contrast to other approaches working at the level of programming in the large. For example, Rist [24] focuses on programming plans, which can involve multiple constructs, and analyses their associated cognitive loads. In another direction, Byckling and Sajaniemi [6] focus on program structure based on the notion of *roles of variables* [26] as a mechanism for understanding the development of knowledge in novice programmers.

This paper discusses how FWEs might be used to provide basic, but effective, programming education. The organization of the paper is as follows. Section 2 provides a review of cognitive load theory, including the CLT memory architecture, the nature of cognitive load, and how CLT impacts pedagogy. Section 3 discusses FWEs in the context of programming education. Sections 4, 5 and 6 provide sample FWEs for selection, iteration, and subroutine call. Section 7 suggests several directions this work could take and touches on the limitations of this approach. Section 8 concludes the paper.

2. COGNITIVE LOAD THEORY

Cognitive load theory (CLT) models the architecture of memory and focuses on how constraints on memory's usage can help determine what kinds of instructional designs and materials are effective [22, 28]. This section reviews cognitive load theory, including the CLT memory architecture, the nature of cognitive load, and how CLT impacts pedagogy.

2.1 Cognitive Architecture

Cognitive load theory is based on a model of memory consisting of three components: *sensory memory*, *working (short-term) memory* and *long-term memory*. A critical element here is that working memory is limited to a small number of slots (less than 10), while long-term memory is effectively unbounded. The model also recognizes three forms of data, two associated with sensory data and one with data stored in long-term memory. Sensory data can be in raw form (in sensory memory) or in an encoded form in working memory. The third form of data is called a *schema*, which is constructed in working memory and stored in long-term memory. Working memory constructs schema by integrating entries from working memory along with a representation for the associations or relationships among the integrated data elements. Over time, schema can become quite abstract, representing a more widely applicable piece of knowledge. Regardless of complexity, however, a schema always takes a *single* slot in working memory, allowing room for additional pieces of information to also reside in working memory.

When new schema are formed we say learning has taken place. However, the schema just formed may be inaccurate or incorrect. If the schema is triggered in the presence of contradictory data, then it must be adjusted or a new more accurate schema must be formed. It is important to remember that learning is itself a cognitive task; learning will be less effective if the context of a problem overburdens working memory.

2.2 Cognitive Load

Cognitive load refers to the load imposed on a learner's working memory; that is, the level of 'mental energy' or working memory capacity that is required to process a given amount of information [20]. Learning occurs in and is directly affected by the load on working memory.

Cognitive load can be generated 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 three kinds of working memory load [20]:*intrinsic* cognitive load is a measure of the inherent difficulty of the material being learned; *germane* cognitive load is a measure of the amount of working memory resources the learner must devote to other data needed for the formation of schema; *extraneous* cognitive load is a measure of the amount of working memory needed by instructional procedures and materials which are not necessary for a learning task.

Intrinsic cognitive load remains unaffected by teaching methods and materials since it is an immutable characteristic of the material being learned. The instructor can, however, affect total load levels by adjusting germane and extraneous cognitive load [20]. While germane cognitive load is useful for the learner, extraneous cognitive load should be kept as low as possible.

2.3 Cognitive Load and Pedagogy

The whole point of cognitive load theory is to provide a framework within which the learning process can be studied. The key issue is to design teaching materials and processes that will lower the cognitive load for students in order to promote effective learning. As suggested in the previous section, if we want to decrease cognitive load, we must focus on germane and extraneous load.

Lowering germane load is appropriate when the problem being addressed has a relatively large number of components. There are two cases. In the first case, the number of components is relatively large because the problem is not sufficiently focused on a particular concept (teaching goal), and the idea is to restate the problem in a simpler more focused form. In the second case, the problem is focused but complex, and there is little to do other than to wait until the learner has mastered topics that can carry some of the cognitive load of the complex problem.

One of the important discoveries of cognitive load theory is that traditional problem solving, i.e., solving problems with specific goals from scratch, is not an effective learning strategy for novices. The problem solving strategy used for such problems is referred to as "means-end analysis", because the student must, at each step, be mindful of the current state of the solution and also the state of the problem goal – keeping track of these two states is a source of cognitive overload. As found by Sweller [28], solving such problems results in solutions to the problems, but not learning.

Since Sweller's findings were published, a considerable amount of research has been carried out in an effort to understand what types of exercises should replace the traditional approach. Applying CLT has allowed researchers to generate new techniques that have produced several positive effects, including the goal free effect, the worked example effect and the partially worked example effect.

The "goal-free effect" is observed when students work problems without specific goals. An example of such a problem is: *determine as many properties as possible of a 3-4-5 triangle using the distance formula and trigonometric identities.* Without a specific goal in mind the student works forward looking for information about the triangle, applying the specified formulas to determine the circumference, area, or height [2]. There is a decrease in the cognitive load because the student works simply with the formulas studied and the description of the triangle — no specific goal has to be kept in working memory.

The "worked example effect" is seen when students study completely worked out examples consisting of a statement of the problem to be solved, a sequence of solution steps for the problem, and a statement of the completed solution. Traditionally, problems are given in pairs: a worked problem followed by an unworked problem, and the student is expected to use the worked problem as a model. Worked examples have been shown to be effective for the acquisition of cognitive skills by novices for near transfer [29, 7].

The "partially worked example effect" has students finish partially completed problem solutions. This technique has been shown to be effective for students studying a wide range of mathematical topics [8, 19] and has been extended to a class of problems called *fading worked examples.*

2.4 Fading Worked Examples

Going from complete worked examples to solving problems from scratch can only be effective if the problems are relatively simple. To extend the strategy, one can make use of partially worked examples that are completed by the student in stages. This approach gives way naturally to the strategy that will be the focus of this section and the rest of the paper – FWEs.

Starting with a completely worked example to serve as a model, an FWE sequence is a sequence of partially worked examples in which each problem in the sequence contains one fewer worked step than its predecessor so that, in the end, the learner is given a problem to solve with no worked steps provided (a solve-from-scratch problem). In forward fading, the first fading problem is missing the first step, the second fading problem is missing the first two steps, and so on. Reverse fading reverses the order.

Fading reduces cognitive load by allowing the learner to focus on a limited number of solution steps at a time, gradually increasing the load as the student acquires declarative knowledge and relevant cognitive skills. It also promotes more efficient learning as less time and effort is needed to attain the same level of transfer ability than with instructional design that does not use fading [11]. As the examples in Sections 4 through 6 will show, the steps outlined for each problem area provide the student with a process for problem solving, thus also providing procedural knowledge.

This approach is most effective for novices when first confronting the material. However, as learners acquire declarative knowledge and cognitive skills, continued use of "plain" FWEs can be counterproductive, producing the "expertise reversal effect" [12]. To also promote far transfer (the ability to apply knowledge to novel scenarios), students must be challenged to do more. For example, process-oriented worked examples have learners justify their solution steps by supplying the principled and strategic information an expert could provide [11].

3. DIMENSIONS OF PROGRAMMING

We propose that by appropriately decomposing the programming process, it is possible to identify smaller components whose cognitive load can be controlled. The programming process is based largely on the appropriate application of sequencing, selection, and subroutine call. But learning these programming structures is more than simply a syntactic endeavor. A critical recognition for our work is that *programming is multi-dimensional* and that a student must gain understanding of the basic algorithmic constructs relative to the critical dimensions of programming. For this reason we focus on these <*construct*, *dimension*> pairs as the basis for our "smaller components."

One dimension of programming is the programming language with its various language structures whose syntax must be mastered: variable, expression, assignment, selection, iteration, etc. A related dimension is the semantics of these constucts. Another is the programming process, beginning with problem analysis and design in the problem domain, continuing with the implementation of the design as a particular programming language statement, and finishing with validation and verification of the resulting code sequence. The challenge for the CS1 student is to master these dimensions (e.g., build effective schema) so that they can be applied correctly and 'effortlessly'.

Our interest is in supporting the student in the development of these schema by focusing their attention on particular dimensions of each construct. The idea is that for each construct, each dimension of interest would have its own sequence of FWEs. Our first step in creating FWEs, then, is to identify and describe the dimensions that guide their construction. Which dimensions are included and their order in an FWE may vary depending on the target construct.

We will focus on the following dimensions of programming.

design: Identify relevant information from the problem description. This may include identifying inputs and outputs, constraints, condition-action pairs, and so on. Using this information, provide a design describing the steps or parts of the solution.

implementation: Translate the design into semantics preserving statements in the target language. If the design has been done well, this dimension may be trivial and could be folded into the design dimension as the last step.

semantics: There are several approaches that can be taken to explore the semantics of a construct.

- **assert:** It has been suggested that far transfer is promoted by having students justify their solution steps [11]. More generally, externalization of the thought process has been shown to help with writing in general [10] and to help science students

better understand and retain what they learn [17]. One way to do this with programming concepts is to follow the implementation FWEs with another set of FWEs in which the student provides assertions about what is known after each programming statement. Following the approach taken in [16], higher level use of assertions should become an integral part of the programming process and one would expect that to occur as a natural outcome of the use of FWEs.

execution: For a set of inputs, indicate which statements from the implementation will execute. This activity has additional meaning when done in conjunction with asserts and verification.

verification: Test oracles can be created from information gathered when developing the design and can be applied after the design has been done and once the implementation is complete. There is a nice synergy between the thought processes needed to create the tests, produce the design, generate assertions, and perform the execution trace.

To develop an FWE sequence for a programming construct, we need to identify sequences of steps for each of the dimensions that will be part of the FWE. As each dimension is presented to the student, it is these steps that will be faded. Significantly, note that providing these steps also gives the student a clear *process* for solving that dimension of the problem. Initially the steps are separated out. For a more experienced programmer this would not only be unnecessary, it would actually slow down the learning process (the expertise reversal effect). But for novices, this level of detail is appropriate. Of course, the goal is that as they gain experience, students will internalize and abstract the detailed steps, making them unnecessary. That successful shift signals movement away from novice status and the creation of useful schemas.

The next three sections illustrate the FWE idea for selection, iteration and subroutine call. As the FWE examples in those sections make clear, the number of combinations possible in constituting an FWE is tremendous (a stark reminder of why students sometimes struggle). A key for the design of FWEs is how to constrain the language problem space to make the presentation tractable (as an FWE and for the student). In this context, it is important to point out that the examples provided here illustrate the possibilities for FWEs and do *not* represent *the* way FWEs for these constructs *must* be done. Instructors can tailor the design of FWEs to their own approach to teaching constructs and the needs (expectations) of their student population (more/less challenging, more/less detailed, more/less variation in the examples, different steps, etc.).

4. FWES FOR SELECTION

Consider the concept of 'selection' from the perspective of design, implementation, semantics, and execution. Handling all of these in a single example will carry a heavy cognitive load. We can control the cognitive load if we focus on the dimensions individually. Here we identify the steps to apply for each of these dimensions. A completed example for each of these dimensions of programming selection follows.

design: In the problem domain the result of designing a selection is a matrix, whose rows correspond to the cases in the selection and whose columns correspond to the conditions and actions associated with the cases. The design steps are as follows:

Step 1: From the problem description, identify the cases of the selection and determine if there is a default case.

Step 2: For each case, determine an appropriate condition; enter it in the appropriate matrix row.

Step 3: For each case, determine the action to be taken when the case is selected. If there is a default, identify the associated action. Enter the actions into the matrix in the corresponding rows.

implementation: The implementation of an appropriate programming statement (in a C-like language) given a design matrix is described as follows.

Step 1: If the first case has a condition $cond_1$ and action $action_1$ write

`if` $(cond_1)$ $action_1$

Step n: For each subsequent case add to the end of the growing statement the following.

`else if` $(cond_n)$ $action_n$

Step last: If there is a default with action $action_d$, add the following at the end of the statement.

`else` $action_d$

semantics - assert: The steps to insert assertions follow quite naturally from the steps for the implementation.

Step: For each `else` in an `else if` $(cond_n)$ $action_n$, provide an `ASSERT` that is the negation of the previous condition, $(cond_{n-1})$, logically ANDed with the `ASSERT` from earlier conditions in the statement.

semantics - execution: Tracing execution of a block of code allows us to focus on the execution semantics of a selection statement. The process begins with a complete selection statement and an initial state, then has the student identify in sequence the statements that are executed.

4.1 Selection Design FWE

Each FWE sequence for a dimension of a construct begins with a fully worked example to serve as a model. All FWEs begin with a description or problem statement that provides a context for the code's development.

Problem: A grade school categorizes its students as being "pre-school" for ages between 4 and 5 (inclusive), and "grade school" for ages between 6 and 11 (inclusive). A program is to input a student's name and age and print a report with the student's name and category. Design an appropriate selection statement that when executed will print the required output. If the student's age does not fit into one of the categories, then an error message will be displayed.

Worked Solution:

Step 1: From the problem description, identify the cases of the selection and determine if there is a default case.

```
    case             condition                  action
pre-school
grade school
default
```

Step 2: Identify a condition for each case.

```
    case             condition                  action
pre-school     4 <= age < 6
grade school   6 <= age < 12
default        age < 4 OR age >= 12
```

Step 3: Identify an action to be completed when the corresponding condition is true.

```
    case            condition                 action
pre-school    4 <= age < 6          print name in 'pre-school'
grade school  6 <= age < 12         print name in 'grade school'
default       age < 4 OR age >= 12  print Error: bad age
```

Discussion What might instruction look like using FWEs? Typically the instructor would walk the students through the steps from some "smaller component" (e.g., selection design), then apply them to a sample problem like the one above. Following the completely worked example, the student would be given a series of *different* problems with faded solutions. The first of these could have Steps 1 and 2 completed, and Step 3 would include the columns for 'case' and 'condition' and the 'actions' column would be empty. The second FWE in the series would only have step 1 completed. The third FWE would have the problem statement followed by the three labelled, but empty, steps. The final FWE could contain just the word problem; the student is asked to remember (or look up, if necessary) the three steps and complete them. This is the solve-from-scratch problem.

As an aside, one can easily see that use of FWEs in the classroom would work beautifully with recent classroom presentation tools such as DyKnow [3] and Classroom Presenter [1] that promote active learning in the classroom by having students ink their copy of the online notes and submit solutions to problems online. Imagine going through a worked example for some construct in class, then introducing the FWEs for the construct. Students generate a solution at their stations and anonymously submit it to the instructor, who selects a submission representing a good solution or perhaps one that highlights a common misconception and uses it to clarify the concepts.

4.2 Selection Implementation FWE

Once the student completes a sequence of FWEs from the design dimension for selection, they would be given a sequence of implementation FWEs for selection. For continuity and to help the student see the process through from problem statement to implementation, the *same set of problem statements are used*. For the implementation dimension, the student would once again be given the problem statement followed by the completed design. The student's task is to translate the design into the target language. For example, the student would first be given a completely worked solution to the worked problem given in Section 4.1 to use as an example. This would be followed by faded versions of the problems they will have seen when doing the faded selection design FWEs. What follows is one of the faded examples.

Problem: Quiet Farms grades eggs by weight: 'small' if weight is between 50 and 60 grams (including 50), 'large' if weight is between 60 and 70 grams (including 60), and 'extra large' if weight is greater than or equal to 70 grams. Under the assumption that only valid weights are supplied (weights >= 50 grams), design and implement a selection statement that will, given an egg's weight, print the weight and classification.

Step 1: Review the action to be completed when the corresponding condition is true.

```
     case        condition           action
small        50 <= weight < 60   print "small"
large        60 <= weight < 70   print "large"
extra large 70 <= weight         print "extra large"
```

Step 2: Using steps provided for implementation of a selection statement and the worked example as a model, complete the translation of the design (selection matrix) into the target language.

```
if ( 50 <= weight && weight < 60 )
  cout << "Egg weight (g): " << weight << " => small";
else if ( _______________ )
  cout << _______________ << endl;
else
   ------------------
```

Discussion In some cases, such as selection, the translation is a straightforward process and could be added as a last step to the design dimension. Instructors would have the option, of course, to split the implementation dimension out to be done separately as a sequence of faded problems, in which case portions of the solution would be elided, to be filled in by the student. One might, for example, include the actions and leave out the conditions, or vice versa, or, as in the example above, remove a mixture of elements.

4.3 Selection Semantics – Assert FWEs

Determining the logical expressions to use for the assertions is a challenging task, but very useful if a student is to really understand the implications of her code. This ability to reason about one's code is also important to debugging, and one would expect it would also positively affect the learning of design skills. The fully worked example below shows the asserts for the problem from section 4.1.

Problem: Members of the sales team received bonuses based on their yearly sales. Sales below $50k receive no bonus, sales in the range of $50k to $75k receive a $500 bonus, sales above $75k and below $100k receive a $750 bonus and sales above $100k receive a $1000 bonus. Write a selection structure to input a salesperson's annual sales and determine the bonus.

```
cin >> yearly_sales;
if (yearly_sales < 50000 )
   bonus = 0;
else  // ASSERT: yearly_sales >= 5000
  if ( yearly_sales >= 50000 && yearly_sales < 75000 )
```

```
  bonus = 500;
else // ASSERT: yearly_sales >= 75000
  if ( yearly_sales >= 75000 && yearly_sales < 100000 )
     bonus = 750;
  else // ASSERT: yearly_sales >= 100000
      if ( yearly_sales >= 100000 )
        bonus = 1000;
```

Discussion An instructor might choose to have students include the ASSERTs while doing the implementation as advocated in [16]. As students develop expertise, one would expect that implementing and asserting would be viewed as synergistic and done at the same time.

Including ASSERTs can be challenging for a student. However, the payoff is promising; after completing the asserted form one can optimize each condition in turn based on the assertion and original condition. In this case, the first two ASSERTs should reveal that the second and third conditions can be shortened, and after the last ASSERT has been placed, the student will hopefully see that the last conditional is not needed and can be safely removed without changing the behavior of the selection statement. These insights might also occur after tracing execution of the code. Letting the student come to this insight on her own (and understanding *why* it works) is a powerful form of learning.

4.4 Selection Semantics – Execution FWEs

In an FWE for execution, the student determines which statements will execute and how state changes. In the context of selection, this means identifying which conditions are evaluated and the action to be executed. The student is given the selection statement (generated while doing the FWEs for implementation) and a table whose columns provide a selection of data values. Given these values, the student identifies which statements will execute and, in some cases, how the program's state changes. Emphasis is on examining all paths through the selection and should include values at the bounds of the condition. The fading examples remove portions of the tracing table. The final, unworked, problem has the student provide both the input values to use for the trace and the trace for those values. The worked example below continues the example from the design FWE presented in Section 4.1.

Worked Solution:

Step 1: Review the selection statement.

```
1.  cin >> age;
2.  if ( 4 <= age && age < 6 )
3.    cout << name << " is in 'pre-school'";
4.  else   // ASSERT: age < 4 OR age >= 6
5.    if ( 6 <= age && age < 12 )
6.      cout << name << " is in 'grade school'";
7.  else   // ASSERT: age < 4 OR age >= 12
8.    if ( age < 4 || age >= 12)
9.      cout << "age entered is in error!";
```

Step 2: Using the worked example as a model, trace execution of the selection statement for the last three columns.

CODE	Values for age				
	3	4	5	6	12
cin >> age;	X	X			
if (4 <= age && age < 6)	X	X			
cout << name << " is in 'pre-school'";		X			
else if (6 <= age && age < 12)					
cout << name << " is in 'grade school'";					
else if (age < 4 || age >= 12)	X				
cout << "age entered is in error!";	X				

Discussion If the student has completed a series of verification FWEs, they will already have identified an appropriate set of test values to exercise the code.

By the time the student has completed the last series of FWEs, they will have experience selection through multiple dimensions and should have a good grasp of the construct. Ideally there is also some metacognitive activity going on and students are seeing the value of having a well-defined process to solve a problem *and* the value of viewing a problem from multiple dimensions.

Lastly, a caution. The selection examples appear trivial to us, but to some novices they will be daunting. One of the hidden challenges for educators is to remember the difficulty we had as students and to realize that what is now obvious (trivial) to us, was once new and puzzling.

5. FWES FOR INDEFINITE LOOPS

While students generally find the *idea* of looping to be straightforward, the details of their design and implementation are often problematic. Aspects posing problems include the kind of loop to use (definite/indefinite; pre/post-test), the loop entry and exit conditions, the notion of a loop control variable (including initialization, use within the loop test, update within the loop, and the relationship between these three subparts), the design of the loop body, and the use of ASSERTs and loop invariants to reason about the loop's correctness.

This section looks at fading worked examples for pre-test, indefinite loops. Separate sets of FWEs would focus on definite loops and post-test indefinite loops. It is not difficult to imagine a follow-up set of problems in which the student must determine what loop type is appropriate and justify that decision through reference to elements of the problem statement. As with the other FWE examples given in this paper, what is shown here is representative of how FWEs can be used to gradually introduce a constellation of related concepts and program components.

design: In addition to the purpose of the loop, the problem statement must contain enough information to determine what type of loop is needed and the loop's starting and terminating conditions. In the problem domain, the result of designing a pretest indefinite loop is a set of initializations, a loop entry/exit condition, and a sequence of steps to execute with each iteration of the loop, including a statement that guarantees the loop makes progress toward termination.

Step 1: What data will the loop use? Identify all the information needed to solve the problem: determine what each item represents within the problem; determine what data need initialization; determine expected values once the loop has terminated.

Step 2: What action is to be repeated? Outline the components of the loop by thinking about how the problem would be solved by hand. From this identify the sequence of steps that need to be repeated.

Step 3: Under what condition is the action repeated? This is called the loop entry condition and the variable(s) that control loop entry are called the loop control variable(s).

Step 4: : What guarantees progress toward termination? Specify a statement that will guarantee progress toward termination.

implementation: Translating the design into an appropriate statement.

Step 1: Provide a skeleton of the while loop. The steps that follow identify the code needed to replace the loop components.

```
variable initialization
loop control variable initialization
while ( loop entry condition )
{
  loop work
  update loop control variable
}
```

Step 2: Choose variables for the data that the loop will need. This includes "inputs" to the loop as well as data the loop will generate as a result. Place these in the variable initialization section that precedes the loop and include a description of their purpose. Provide each variable with an initial value where it is known.

Step 3: Write the set of statements identified in Step 2 of Design as the body of the loop. Place these in the loop body as "loop work".

Step 4: Create the Boolean expression for the loop entry condition and place it in the "loop entry condition" portion of the loop. Provide statement(s) to initialize the loop control variable(s), paying particular attention to the interaction of the initial value of the loop control variable(s), the test done on the loop control variable(s) in the loop entry condition, and the expected value of the loop control variable(s) on exit from the loop.

Step 5: Add a statement to the loop body that guarantees progress toward termination. Where this appears in the loop body is determined by how the loop control variable is initialized and is used in the loop.

semantics - assert: There are three obvious points in the implementation to insert assertions.

Step 1: Insert `ASSERT`s before the loop entry point stating what is known about the values of all loop variables prior to the loop's execution.

Step 2: After each iteration of the loop, insert an `ASSERT` to confirm the values of all variables before the next iteration.

Step 3: Provide `ASSERT`s following the loop indicating what is known about the values of all loop variables on exit from the loop.

semantics - execution: Tracing through the loop with a pre-determined set of values will help the student see both the action performed in the loop as well as the interplay of the initialization, use, and update of the loop control variable.

5.1 Indefinite Iteration Design FWE

As with all FWE sequences, the student is first given a completely worked solution as a model for the process. Subsequent FWEs in the sequence leave out individual steps.

Problem: Retrieve a set of grades from the user via the keyboard, computing their sum and keeping track of the number of grades entered. A later part of the program will perform a calculation of the average of the grades entered, which requires a total of all the grades and the number of grades entered. Valid grades are integers in the range of 0 to 100 inclusive and all input is assumed to be valid. The end of data will be indicated by entering -1.

Worked Solution:

Step 1: What data will the loop use and how should it be initialized?

```
sum ← 0      // the sum of all grades; no grades
             //   have been added to the total yet
count ← 0    // the count of grades read in; no grades
             //   have been read in yet
grade        // the loop control variable
prompt for first grade
grade ← read in first grade    // init grade from input
```

This is a sentinel-controlled loop: -1 is the sentinel value.

Step 2: What action is to be repeated to solve the problem?

```
sum ← sum + grade    // add new grade to running sum
count ← count + 1    // one more grade has been read
```

Step 3: Under what condition is the action repeated?

As each grade is read, it is added to sum and count is incremented by 1. This stops when there is no more data; that is, when a grade of -1 is read in: grade != -1.

Step 4: What guarantees progress toward termination?

The input statement before the loop begins reads in the first grade. If the loop is entered, that grade is treated as valid and must be processed. The first few statements of the loop process the last valid grade read in. The last action of the loop is to read in the *next* value to process, which may be a grade or the sentinel value -1.

```
prompt for grade            // ask for the next grade
grade ← read next grade     // read next grade; update
                            // the loop control variable
```

Discussion: The placement of the statements for obtaining the values for each grade can be difficult for beginning students to grasp. The first value for the grade needs to be obtained prior to the loop's start; and then a new value for grade must be retrieved prior to the completion of the loop's action. This two-step initialization and update of the loop control variable is often troublesome for students. Asking a student to narrate their decisions as is done in the example above helps them, first, to see the interrelationships, and, second, to reason about them.

Also note the heavy use of commenting to describe the information items used in the loop and the statements that manipulate them.

Finally, if the students have been through selection, a natural extension to this problem is to state that invalid grades are to be rejected, requiring a selection statement to be a part of the loop body.

5.2 Indefinite Iteration: A Faded Design Example

What follows is an example of fading for iteration design. The final step is left to the student to solve.

Problem: A local bank offers a savings account with a 5% interest rate that is compounded annually. That is, if you place $1,000 in the account, at the end of the year the account will have accrued $50 interest, making the balance $1,050. At the end of the second year, the account will have accrued an additional $52.50, producing a balance of $1,102.50. With an initial deposit of $1,000 you want to know how many years it will take for the balance on the account to reach or exceed $10,000.

Step 1: What data will the loop use and how should it be initialized?

```
balance ← 1000    // the account balance
goal ← 10000      // the goal for the account balance
years ← 0         // the number of years to the goal
                  // balance; no years have passed yet
interest ← 0      // initially there is no interest
```

Step 2: What action is to be repeated to solve the problem?

Each iteration of the loop represents passage of a year. Compute the interest on the current balance and add it to the balance; update the number of years that have passed.

```
interest ← balance * .05          // compute interest
balance ← balance + interest      // balance with new interest
years ← years + 1                 // a year has passed
```

Step 3: Under what condition is the action repeated?

The loop continues until the balance on the account reaches or exceeds the goal balance: balance <= goal

Step 4: What guarantees progress toward termination?

The student completes this step

Discussion In this case, the work to be done in the final step was already done in an earlier step; something the student will have to identify. Subsequent FWEs for iteration design could leave out some or all of Step 2, then Step 1.

5.3 Indefinite Iteration Implementation FWE

This section continues the problem from Section 5.1, illustrating the process the student would use to complete the implementation based on the design. This is the fully worked example students would be given as a model. We follow this with a faded example.

Problem: See problem statement in Section 5.1.

Step 1: Provide a skeleton of the while loop.

```
variable initialization
loop control variable initialization
while ( loop entry condition )
{
  loop work
  update loop control variable
}
```

Step 2: Choose variables for the data the loop will need and place these in the variable initialization section with a description of their role within the loop. Provide each variable with an initial value where it is known.

```
int count = 0;   // the count of grades read in; no
                 //    grades have been read in yet
int sum = 0;     // sum of all grades; no grades have
                 //   been added to the total yet
int grade;       // loop control variable
loop control variable initialization
while ( loop entry condition )
{
  loop work
  update loop control variable
}
```

Step 3: Write the set of statements identified in Step 2 of Design as the body of the loop. Place these in the loop body as "loop work".

```
int count = 0;   // the count of grades read in; no
                 //    grades have been read in yet
int sum = 0;     // sum of all grades; no grades have
                 //   been added to the total yet
int grade;       // loop control variable
loop control variable initialization
while ( loop entry condition )
{
  sum = sum + grade; // add new grade to running sum
  count = count + 1; // one more grade has been read
  update loop control variable
}
```

Step 4: Create the Boolean expression for the loop entry condition and place it in the "loop entry condition" portion of the loop. Provide statement(s) to initialize the loop control variable(s), paying particular attention to the interaction of the initial value of the loop control variable(s), the test done on the loop control variable(s) in the loop entry condition, and the expected value of the loop control variable(s) on exit from the loop.

```
int count = 0;   // the count of grades read in; no
                 //    grades have been read in yet
int sum = 0;     // sum of all grades; no grades have
                 //   been added to the total yet
int grade;       // loop control variable

cout << "Enter first grade: ";
cin >> grade;    // get 1st grade from the keyboard
while ( grade != -1 )
{
  sum = sum + grade; // add new grade to running sum
  count = count + 1; // one more grade has been read
  update loop control variable
}
```

Step 5: Add a statement to the loop body that guarantees progress toward termination.

```
int count = 0;   // the count of grades read in; no
                 //    grades have been read in yet
int sum = 0;     // sum of all grades; no grades have
                 //   been added to the total yet
int grade;       // loop control variable

cout << "Enter first grade: ";
cin >> grade;    // get 1st grade from the keyboard
while ( grade != -1 )
{
   sum = sum + grade;   // add new grade to running sum
   count = count + 1;   // another grade has been read
   cout << "Enter next grade: ";
   cin >> grade;        // update loop control variable
}
```

Discussion: An instructor may choose at this point, or during use of a semantic-assertion FWE, to explain the flaw in the following code.

```
int counter = 0;
int total = 0;
int grade = 0;

while ( grade != -1 )
{
  cin >> grade;
  total = total + grade;
  counter++;
}
```

Presentation of flawed solutions such as this during the use of the FWEs provides the students with a concrete experience of the benefit of using all dimensions of program development.

5.4 Indefinite Iteration: A Faded Implementation Example

This section continues the faded problem from Section 5.2, illustrating the use of fading for the implementation.

Problem: See problem statement in Section 5.2.

Step 1: Provide a skeleton of the while loop.

```
variable initialization
loop control variable initialization
while ( loop entry condition )
{
  loop work
  update loop control variable
}
```

Step 2: Choose variables for the data the loop will need and place these in the variable initialization section with a description of their role within the loop. Provide each variable with an initial value where it is known.

```
float balance = 1000;         // the account balance
float goal_balance = 10000;   // goal account balance
int years = 0;                // # of years to goal balance;
                              // no years have passed yet
float interest = 0;           // initially there is no interest

loop control variable initialization
while ( loop entry condition )
{
  loop work
  update loop control variable
}
```

Step 3: Write the set of statements identified in Step 2 of Design as the body of the loop. Place these in the loop body as "loop work".

```
float balance = 1000;         // the account balance
float goal_balance = 10000;   // goal account balance
int years = 0;                // # of years to goal balance;
                              // no years have passed yet
float interest = 0;           // initially there is no interest
loop control variable initialization
while ( loop entry condition )
{
  interest = balance * .05; // compute interest accrued
  balance = balance + interest; // balance with new interest
  years = years + 1; // a year has passed
  update loop control variable
}
```

Step 4: Create the Boolean expression for the loop entry condition and place it in the "loop entry condition" portion of the loop. Provide statement(s) to initialize the loop control variable(s), paying particular attention to the interaction of the initial value of the loop control variable(s), the test done on the loop control variable(s) in the loop entry condition, and the expected value of the loop control variable(s) on exit from the loop.

The student completes this step

6. FWES FOR SUBROUTINE CALL

Among the issues many students find troublesome with subroutine calls are: when to pass a parameter, what to pass as a parameter, the parameter passing mechanisms, type compatibility of formal and actual parameters, constraints on parameters (e.g., greater than 0, not null, not empty) where a method may be called, what to do with the return value, and how the call changes the program's/object's state. Here we describe two dimensions of subroutine calls, followed by portions of FWEs for them. The descriptions assume a C-like language in which the subroutine call is self-contained (i.e., all the data it needs is either in the parameter list or the subroutine body). Descriptions of these

dimensions would be slightly different if we were considering methods defined as parts of classes (most notably the issue of a method accessing state information). Each FWE problem statement includes a behavioral description of the subroutine, including its inputs and outputs.

design: In the problem domain the result of designing a subroutine call is a list of characteristics of the inputs and outputs. The design steps are as follows.

Step 1: Parameters – examine the subroutine behavioral description to determine

- the number of parameters expected
- the type and order of each parameter and what each parameter *represents*
- the parameter passing mechanism used for each parameter and whether it is an IN,OUT or IN/OUT parameter.

Step 2: Return value – examine the subroutine behavioral description to determine what type of data, if any, is returned by the subroutine and what it represents.

implementation: The implementation of an appropriate subroutine call in a C-like language given the list of parameters and return characteristics is as follows.

Step 1: If there is a return value, determine how it will be used (in an assignment, part of an expression or output statement, etc.).

Step 2: Identify the actual parameters that need to be passed to the subroutine to solve the problem given.

Step 3: Determine the mapping of actual parameters to formal parameters (e.g., identify the actual parameter that will map to the first formal parameter, and repeat for the other parameters).

Step 4: Using the information gathered in Steps 1 through 3, write an appropriate subroutine call.

6.1 Subroutine Call Design

As with selection and iteration, the student is first given a completely worked problem to be followed by a series of faded problems. Following is a completely worked example.

Problem: Given the following subroutine description, write C++ code to output "found" if a search value `letter` is found in char array `name` containing `n` elements, otherwise output "not found".

```
// Search for a target value in an array.
// Input: values - a char array (IN)
//        size - the number of elements in values (IN)
//        target - a char to search for in values (IN)
// Output: return the array index of the first occurrence
//         of target if found, -1 otherwise
int search( const char values[], int size, char target );
```

Worked Solution:

Step 1: Gather information about the formal parameters

- Number of formal parameters: 3
- Type, order and meaning of the formal parameters:
  1. char[] – IN; the char array to search
  2. int – IN; the number of cells in the array containing data to examine
  3. char – IN; the character to search for in the input array
- Parameter passing mechanism for each formal parameter:
  1. char[] – const pass-by-reference
  2. int – pass-by-value
  3. char – pass-by-value

Step 2: Gather information about the subroutine's return value

A search has two possible results: success and failure. If successful, `search()` returns the integer index where the `target` can be found in the input array. An unsuccessful search means that `target` was not found in `values` in the cells indexed from 0 to `size` - 1 inclusive; an unsuccessful search is indicated by returning -1.

Discussion The completely worked example addresses several basic problems novices have with design of subroutine calls. As the example illustrates, the design process forces the student to think very explicitly about the elements of a subroutine call *and* their meaning within the context of the problem. The value of writing out the answers to each step is that it encourages (but, alas, cannot guarantee!) clarity in the student's thinking.

FWEs should be designed for "teachable moments". For example, in the problem above, one could talk about the significance of the `const` with the array parameter. How does that affect the subroutine call? If it doesn't affect the call in this case, why is it there? This discussion should have a carry over effect to subroutine design.

6.2 Subroutine Call: A Faded Design Example

Problem: Given the following subroutine description, write C++ code to output the balance of a bank account that has annual compound interest after `n` years, with initial balance `balance`, and an interest rate of `4%`.

```
// Compute the balance assuming an interest rate of
//   rate for the specified number of years.
// Input: balance - initial account balance (IN)
//        rate - annual interest rate (IN)
//        years - # of years to accrue interest (IN)
// Output: new balance with compound interest
float calc_compound_interest( float balance, float rate,
                              int years );
```

Worked Solution:

Step 1: Gather information about the formal parameters

- Number of formal parameters: 3
- Type, order and meaning of the formal parameters:
  1. float – IN; the initial bank balance
  2. float – IN; the annual interest rate applied to the balance

3. int – IN; the number of years to compute interest for

- Parameter passing mechanism for each formal parameter:

1. float – pass-by-value
2. float – pass-by-value
3. int – pass-by-value

Step 2: Gather information about the subroutine's return value

The student completes this step

Discussion The faded problems first leave out Step 2. Subsequent faded problems could leave out all of Step 1 or some of its substeps. As with all FWEs, the goal is that through gradual exposure, repetition and variation, students will come to know the components of each construct and the issues they raise, and will internalize the processes involved in solving problems with the construct.

6.3 Subroutine Call Implementation

Problem: See problem statement in Section 6.1.

Step 1: Determine how the return value will be used.

The problem statement indicates that one of two strings is to be output depending on whether the search value is found. We can make use of the selection matrix here.

```
  case         condition                  action
found   search(name, n, letter) != -1  output 'found'
!found  search(name, n, letter) == -1  output 'not found'
```

Step 2: Identify actual parameters to be used in the subroutine call.

This information is found in the problem statement. The input array is `name`, the number of elements in `values` is `n`, and the char to search for is `letter`.

Step 3: Map the actual parameters to their corresponding formal parameters.

```
1  name    ⇒  values
2  n       ⇒  size
3  letter  ⇒  target
```

Step 4: Using the information gathered in Steps 1 through 3, write an appropriate subroutine call.

```
if ( search(name, n, letter) != -1 )
   cout << ''found'' << endl;
else
   cout << ''not found'' << endl;
```

Discussion Note the opportunities for variation here. For example, having the return value used in an expression, as part of an assignment statement, as an argument to another subroutine call, and so on.

7. DISCUSSION AND FUTURE WORK

As pointed out in the Introduction, the evidence that students find learning to program difficult has been well-documented. The case for taking cognitive load into account is compelling. Certainly there is an intuitive appeal to the idea of breaking the programming process into its different dimensions, as we do with FWEs. Indeed, one of the surprising results of this project for us has been (re-)realizing the number of and variation in the "pieces" that go into the constructs we expect our students to master.

Of course, the decomposition idea is not new. What FWEs add to this approach is *graduated* and *repeated* exposure to the dimensions of programming through working a *variety* of problems in an area. Graduated exposure addresses the cognitive load question. Solving from partially worked problems helps develop near transfer skills. Repeated exposure across a variety of problems and having students justify their solution steps help develop far transfer skills, which are essential to building effective schema. So, the approach sounds promising, but the big question, of course, is will FWEs actually improve student programming skills in an introductory programming course. We will be looking for partners to test this hypothesis by deploying FWEs in their early programming courses.

The FWEs we have developed can easily be redone for other languages that include selection, iteration, subroutines, etc. We have not investigated the possibility of developing FWEs for, say, functional languages. Nor are we certain that we have identified the bounds for what can profitably be presented using FWEs. What aspects of a language aren't amenable to using FWEs? For example, can we use FWE's to present inheritance and polymorphism? Another possibility is looking into extending the use of FWEs for other dimensions of programming such as documentation, efficiency, and programming style.

The question about whether FWEs can also promote general problem solving ability is interesting. There has been some discussion about whether or not the problem solving skills developed through programming instruction are transferrable beyond the programming domain [21]. If FWEs help students to see the value of developing a process to solve a problem, then even if the particular steps may not be transferrable, the idea of identifying a process might be.

8. CONCLUSION

Learning programming concepts involves developing a variety of cognitive skills which decades of experience have shown are challenging for students. Cognitive load theory tells us that overloading working memory inhibits learning. Fading worked examples have emerged from cognitive load theory as an effective strategy for lowering cognitive load by providing gradual exposure to programming concepts. Developing effective FWEs for programming requires thinking clearly about the concepts to be presented, decomposing them into their basic parts, and presenting those basic parts in terms of the multiple dimensions of programming.

Our suggested use of FWEs is limited to individual constructs - discrete blocks of code. It does not (cannot?) fully address the larger problem of program design. The belief, however, is that a deeper understanding of the design, semantics and testing of program parts will facilitate development of higher level program skills.

9. REFERENCES

[1] Richard Anderson, Ruth Anderson, K. M. Davis, Natalie Linnell, Craig Prince, and Valentin Razmov. Supporting active learning and example based instruction with classroom technology. In *SIGCSE '07: Proceedings of the 38th SIGCSE technical symposium on Computer science education*, pages 69–73, New York, NY, USA, 2007. ACM Press.

[2] P. Ayres. Why goal free problems can facilitate learning. *Contemporary Educational Psychology*, 18:376–381, 1993.

[3] Dave Berque. An evaluation of a broad deployment of dyknow software to support note taking and interaction using pen-based computers. *J. Comput. Small Coll.*, 21(6):204–216, 2006.

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

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

[6] Pauli Byckling and Jorma Sajaniemi. A role-based analysis model for the evaluation of novices' programming knowledge development. In *ICER '06: Proceedings of the 2006 international workshop on Computing education research*, pages 85–96, New York, NY, USA, 2006. ACM Press.

[7] R. Catrambone. Generalizing solution procedures learned from examples. *Experimental Psychology: Learning, Memory and Cognition*, 22:1020–1031, 1996.

[8] G. Cooper and J. Sweller. Effects of schema acquisition and rule automation on mathematical problem-solving transfer. *Journal of Educational Psychology*, 79(4):347–362, 1987.

[9] 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.

[10] Janet Emig. Writing as a mode of learning. *College Composition and Communication*, 2(2):122–128, 1977.

[11] T. Van Gog, F. Paas, and J.J.G. Van Merriënboer. Process-oriented worked examples: Improving transfer performance through enhanced understanding. *Instructional Science*, 32(1-2):83–98, 2004.

[12] S. Kalyuga, P. Ayres, P. Chandler, and J. Sweller. The expertise reversal effect. *Cognition and Instruction*, 38:23–32, 2003.

[13] R. Lister, E. S. Adams, S. Fitzgerald, W. Fone, J. Hamer, M. Lindholm, R. McCartney, J. E. Moström, K. Sanders, O. Seppälä, B. Simon, and L. Thomas. A multi-national study of reading and tracing skills in novice programmers. In *Working group reports from ITiCSE-2004*, New York, NY, 2004.

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

[15] M. McCracken, V. Almstrum, D. Diaz, M. Guzdial, D. Hagan, Y. B.-D. Kolikant, C. Laxer, L. Thomas, Ian Utting, and Tadeusz Wilusz. A multi-national, multi-institutional study of assessment of programming skills of first-year CS students. In *Working group reports from ITiCSE-2001*, pages 125–180, New York, NY, USA, 2001. ACM Press.

[16] J. Mead and A. Shende. *Persuasive Programming*. Franklin, Beedle, & Assoc., Wilsonville, OR, 2001.

[17] Paul Miller. *Composition and Technology: A Pragmatic Approach*. PhD thesis, Ohio State University, 1999.

[18] R. Moreno, M. Reisslein, and G. M. Delgoda. Toward a fundamental understanding of worked example instruction: Impact of means-ends practice, backward/forward fading, and adaptivity. In *FIE '06: Proceedings of the 36th Frontiers in Education Conference*, 2006.

[19] F. Paas and J. Van Merrienboer. Variability of worked examples and transfer of geometrical problem-solving skills: A cognitive-load approach. *Journal of Educational Psychology*, 86(1):122–133, 1994.

[20] F. Paas, A. Renkl, and J. Sweller. Cognitive load theory and instructional design: Recent developments. *Educational Psychologist*, 38(1), 2003.

[21] D. B. Palumbo. Programming language/problem-solving research: A review of relevant issues. *Review of Educational Research*, 60(1):65–89, 1990.

[22] A. Renkl and R. Atkinson. Structure the transition from example study to problem solving in cognitive skill acquisition: A cognitive load perspsective. *Educational Psychologist*, 38(1):15–22, 2003.

[23] A. Renkl, R.K. Atkinson, and C.S. Grosse. How fading worked solution steps works – a cognitive load perspective. *Instructional Science*, pages 59–82, 2004.

[24] Robert S. Rist. Schema creation in programming. *Cognitive Science*, 13:389–414, 1989.

[25] A. Robins, J. Rountree, and N. Rountree. Learning and teaching programming: A review and discussion. *Computer Science Education*, 13(2):137–172, 2003.

[26] J. Sajaniemi. Visualizing roles of variables to novice programmers. In *Proceedings of the Fourteenth Annual Workshop of the PPIG*, pages 111–127, 2002.

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

[28] J. Sweller. Cognitive load during problem solving: Effects on learning. *Cognitive Science*, 12(2), 1988.

[29] J. Sweller and G. A. Cooper. The use of worked examples as a substitute for problem solving in algebra. *Cognition and Instruction*, 2(1):59–89, 1985.

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

[31] K. VanLehn. Cognitive skill acquisition. *Annual Review of Psychology*, 47:513–539, 1996.

[32] L. E. Winslow. Programming pedagogy — a psychological overview. *SIGCSE Bulletin*, 28(3), 1996.