# Design Early Considered Harmful: Graduated Exposure to Complexity and Structure Based on Levels of Cognitive Development

Duane Buck
*Otterbein University*, DBuck@otterbein.edu

David J. Stucki
*Otterbein University*, dstucki@otterbein.edu

# Design Early Considered Harmful: Graduated Exposure to Complexity and Structure Based on Levels of Cognitive Development

**Duane Buck and David J. Stucki**
**Otterbein College**
**Mathematical Sciences Department**
**Westerville, OH 43081**
**{DBuck, DStucki} @otterbein.edu**

## Abstract

We have recognized that the natural tendency to teach according to the structure of one's own understanding runs contrary to established models of cognitive development. Bloom's Taxonomy has provided a basis for establishing a more efficacious pedagogy. Emphasizing a hierarchical progression of skill sets and gradual learning through example, our approach advocates teaching software development from the inside/out rather than beginning with either console apps or monolithic designs.

## Keywords

Inside/out Pedagogy, CS1, CS2, Control Structure Diagrams, Bloom's Taxonomy, Formal Specifications.

## 1. Introduction

The traditional liberal arts (arising out of the classical quadrivium and trivium) divided education and the instruction of knowledge into a hierarchy. There was a prescribed order to the subjects, based on their dependency relationships. Language grammar was seen as a prerequisite to logic, which in turn was a necessary precursor to rhetoric. In other words, it has long been obvious to educators that students must master the basics before attempting more advanced, abstract endeavors. Somehow, in the fury of technological advancement we have lost sight of this in computer science education.

Recent attempts to embrace the object-oriented paradigm in CS1 (at least in many textbooks) have resulted in students being exposed to very complex and often subtle concepts before they have any adequate contextual foundation upon which to base comprehension. This problem isn't new. Even earlier approaches to CS1 had students designing entire applications from the start (albeit small ones first). We feel strongly that students learn better when they are provided a context that constrains their thinking in a directed fashion. In other words, expecting them to program to specifications is a way of providing guidance and mentoring without having to give them cookbook instructions.

***Proceedings ACM SIGCSE Symposium, March 8-12, 2000, Austin, Texas.***

## 2. A New Pedagogy

In this paper, we argue that practitioners of computer science education have much to learn from extant research on educational pedagogy. We draw specifically from cognitive development theory and the pedagogy of teaching writing. This has led us to an approach for early computer science education that is at odds with existing textbooks[1].

### 2.1 Analogy to Composition

Our premise is that students learn best when they are given a chance to learn building blocks before they are asked to design the whole building. In teaching writing (at least outside the U.S.), one starts with writing sentences, then paragraphs, then essays. In computer science, an error has been made by assuming that the student should start out by writing the equivalent of a whole literary form. We don't know exactly how this misguided approach got started, but we present below a few possible influences.

In teaching writing, practitioners must first imagine, and then specify the context of a paragraph or essay fragment prior to assigning the writing of it. One motivation for requiring a whole form in computer science is that computers are very narrow minded and require precise adherence to a certain form. With some advance preparation, however, we can avoid forcing the student to write whole forms, but only require that they complete a missing fragment in a more complete work, where they are told the "message" the fragment is to convey. In other words, learn to program from the inside out, solving smaller problems first.

### 2.2 Cognitive Development

We have also observed that Bloom's taxonomy of cognitive learning is helpful in structuring the beginning computer science curriculum. Each level in the hierarchy is subsumed by the next level, so that higher order functioning requires by necessity the lower level skills. In Figure 1 we list each level and the kinds of behaviors that might be expected of a computer science student operating at that level.

[1] In response, we have developed a set of web-based materials to support our methods, and are making the materials freely available to others. (See http://math.otterbein.edu/sigcse/)

| **Cognitive Level**[2] | | **Activity related to CS** |
|---|---|---|
| Knowledge | The remembering of previously learned material. This may involve recall of a wide range of material, from specific facts to complete theories, but all that is required is bringing to mind the appropriate information. | Mathematical pre-requisites; exposure to simple, standard libraries; instruction in syntactic and semantic fluency in a programming language. |
| Comprehension | The ability to grasp the meaning of material. This may be shown by translating material from one form to another (words to numbers), by interpreting material (explaining or summarizing), or by estimating future trends (predicting consequences or effects). | Mental simulation of interpreter: Predict the control flow through a program line by line as it executes. Read a program and predict its results. Translate a program to a flowchart. |
| Application | The ability to use learned material in new and concrete situations. This may include the application of such things as rules, methods, concepts, principles, laws, and theories. | Implement a method to satisfy a specification. Use of library components. |
| Analysis | The ability to break down material into its component parts so that its organizational structure may be understood. This may include the identification of parts, analysis of the relationship between parts, and recognition of the organizational principles involved. | Read and comprehend a system with the objective of making a modification in functionality. Performance analysis of algorithms. Debugging. |
| Synthesis | The ability to put parts together to form a new whole. This may involve the production of a unique communication (theme or speech), a plan of operations (research proposal), or a set of abstract relations (scheme for classifying information). Stresses creative behaviors, with major emphasis on the formulation of new patterns or structure. | Write an ADT, including developing the API. Design of an application, given the requirements. |
| Evaluation | The ability to judge the value of material (statement, novel, poem, research report) for a given purpose. The judgments are to be based on definite criteria. These may be internal criteria (organization) or external criteria (relevance to the purpose), which may be determined by the student or be provided. | Systems analysis. Evaluation of disparate inputs (management, users, systems personnel, etc.) to form the requirements for a single coherent system to meet the needs of the organization. |

**Figure 1: Bloom's Taxonomy Applied to Computer Science Education**

When we teach, we have an inclination to recapitulate the systems development process, because that is the order in which we have learned to apply our craft. We see here that cognitive development corresponds, more or less, to the *reverse* of the ordering of activities in the usual systems development process. So our inclination is to present topics in an order that is dissonant with the cognitive development of our students. For some students, whose cognitive development is already advanced, this may be appropriate. However, with the increasing diversity of backgrounds of students selecting computer science as a major, we no longer have the luxury of ignoring pedagogical issues.

It's not that we have been lazy; we have really been too heroic. Some disciplines are comfortable with the knowledge and comprehension levels for too long a period of time, well past the time that their students should be developmentally ready to move on. They may be able to fool themselves into believing that their students are exhibiting the higher levels of development. However, because our students create artifacts that have tangible, independent evaluation (the compiler and run-time performance), we have a built-in reality-check. This independent evaluation may be another factor encouraging us to push our students beyond their developmental levels. We should not be afraid to honor our students' developmental status: if we give assignments that are consonant with their developmental level, we will obtain *better* outcomes.

[2] Cognitive level descriptions adapted from [1], pp. 506-507.

### 2.3 Inside/out : Upside/down

Based on these pedagogical influences, the process of learning software engineering should turn the Development Life Cycle on its head, incrementally building towards design and then analysis.

Teaching procedures early [8] misses the whole point. Procedures are a way to *structure* the solution of a problem. How can students try to structure the solution, before they have any idea of what a solution is? Read/call early [8] is also a strange approach. Here we are dealing with huge, complex primitives, with many possible behaviors. For example, is it reasonable to expect students early in CS1 to wade through the Java™ API (application programmer's interface) specification to locate the appropriate tools for solving some specified problem? Hardly! Principled utilization of commercial APIs requires sophisticated synthesis of components that in turn must be thoroughly understood (comprehended).

We may appear too reductionistic, but we have a limit. The limit is using the primitives of the language being taught. Why not go to a lower level and teach assembly language first? Or machine language? The answer is that the primitives of a procedural language reveal enough of the underlying mechanism to get a feel for it, and the vast majority of programmers never have to deal with that level of detail. On the other hand, as we explain below, we are experimenting with classical flowcharts and programming without expressions as exercises to increase student comprehension of the underlying processes.

So aren't we just going back to the old way of teaching programming? Not really. The old way had the student doing too much (writing a whole, monolithic application), things that are really beyond them at that point in cognitive development. That method induced a sense of panic, and really taught bad design habits. We try to introduce one new idea at a time, with the student both being motivated and having the background to comprehend and apply it. This incremental approach provides a more graduated learning path.

Inside/out puts the student in the context of an overall application design in which students, while working on mastering one level, can glimpse the more advanced concepts present in the layered interface. They are asked to design and code algorithms, first using only language primitives and later some simple library calls. In the context of the design, they are given complete specifications to which their algorithms must conform. We feel that the specifications are important. Our method of specification usually consists of a formal (mathematically rigorous) part followed by an informal restatement.

We have used the inside/out approach in both procedural and object oriented (OO) designs, and it works well in both. However, it seems to have a real synergy with an object-oriented language, like Java™. We strive to make the design of our assignments reasonably close to the way a non-academic (realistic) design might look. OO designs often utilize methods that simply modify the object state. These are perfect for the student to implement. There are no input/output issues for the student, no parameter passing, no return value, yet it is a reasonable design.

In other words, we are not teaching an OO language procedurally forming many bad habits on the way; from day one the student sees and implements parts of good object oriented designs. As we progress to more advanced levels, the student moves through the following roles, establishing a progression of skill sets:

- server/client of language primitives
- server/client of simple libraries
- server/client of ADTs
- server of ADTs/client of ADTs
- server with application model responsibility
- server of people (application)/client of OS functionality

## 3. Pedagogy In Practice

### 3.1 Developing Inside/Out Assignments

Our assignments typically focus on a single topic, and supply the student with the surrounding environment to explore that topic. The application typically includes a nicely designed graphical user interface. However, because of the careful layering of the design, we also may invoke the student's solution through a batch process in order to evaluate its conformance to the specification. An advantage to the approach is that the student gets the feeling that the user is in charge, instead of some weird question-and-answer user interface that many textbooks proffer, and also that the same functionality may be invoked through different avenues, given a properly layered design.

Our first teaching language was Delphi™ Pascal, which had a reasonably good IDE (integrated development environment) for GUI (graphical user interface) development. However, we are now switching to Java™, and so far we have been coding the assignments in straight Swing code. As the IDEs for Java™ mature, the development of assignments should become easier. The hardest part is coming up with the API and the specifications for the methods that the student is required to implement. One thing to avoid is having the student work directly within the user interface code. They should work in supporting class file(s), and observe the delegation of responsibilities present in a good design. For instance, direct coding into visual forms (provided by IDEs such as Visual Basic™ and JBuilder™) should be avoided. Seeing these unscalable designs is harmful to the student's development of design intuition during their most formative year.

We have been teaching Java using GRASP, from Auburn University. It annotates the code with a Control Structure Diagram (CSD), which gives an intuitive visual indication of what the code means. See Figures 3 and 4. (It also annotates ADA95, C, and C++ in the same style). Far from being a generic pretty-printer application, we have found it an important tool for helping our students better internalize the meanings of control structures. (We also heavily use it in our own development efforts.) We think CSDs are especially useful for supporting student comprehension of Java's strange C based programming syntax. Those curly braces and saying `void` or `int` to create procedures or functions is hard to comprehend. Fortunately, GRASP solves the problem by analyzing the syntax and annotating the source code with graphic symbols evocative of the dynamic behavior of the program.

An encouraging discovery for us was the BlueJ [3, 4] project at Monash University. BlueJ provides support for inside-out teaching and assignments (using Java™) with much less up-front effort on the part of the faculty member. After developing the API that the student will work to implement, the BlueJ GUI environment provides direct support for the student exercising his or her implementation, without having to code a GUI. The student can code and see the behavior of the inside, without there even being an explicit outside! We have had limited classroom experience with BlueJ, but the results are so far encouraging. Another use of BlueJ as a pedagogical tool is to allow a student to learn an ADT by interactively exploring the API (in a bottom-up fashion). This puts the student in a less abstract role than that of *programming* using the ADT. This is consistent with Bloom's notion of comprehension preceding application in the learning process. It is especially useful for visualizing linked structures that cooperate to carry out a specification. We still believe that custom GUIs designed by faculty for an assignment are important for giving the students a feeling for layered design, especially at the earliest stages. Our hope is that BlueJ will support us in latter CS1 and CS2 projects, after our students have learned to visualize the interactions that go on in a complete application.

### 3.2 Example

Our CS1 course starts with a brief introduction to programming with Jarel the Robot, a Java-based derivative of Pattis's [9] Karel the Robot. Because the Jarel language has no explicit variables, we are better able to focus the student on the sequence, selection, and iteration constructs. The environment simulates a task with which they are already familiar: moving around in the world. In fact, they often visualize themselves as Jarel when they are developing their algorithms. Although we eschew design early in general, we are not opposed to introducing some design elements when the student is both motivated and cognitively ready. We believe this is the case toward the end of the Jarel section, where we introduce defining new instructions, starting with the classic "turnright" implemented with a sequence of three primitive turnleft instructions. The student's desire for symmetry is a fantastic motivator!

We then proceed to introduce the Java™ language by way of assignment, variables, and sequential flow. After a first, simple project (the classical Fahrenheit to Celsius problem), and a review of selection constructs, the second project asks the student to implement two methods in a quadratic equation tutoring application.

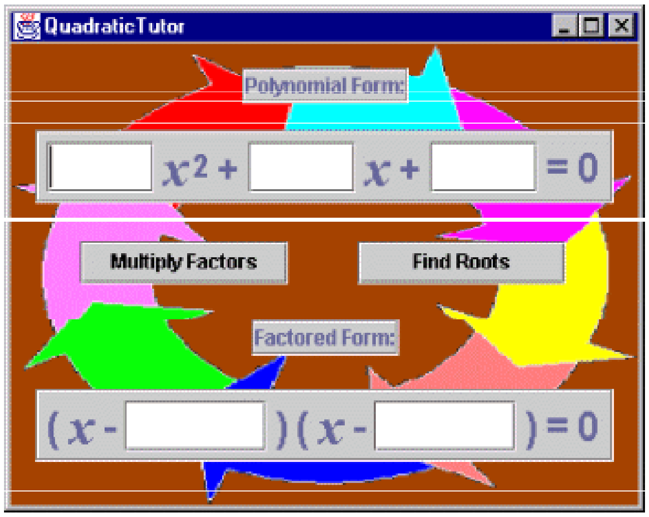
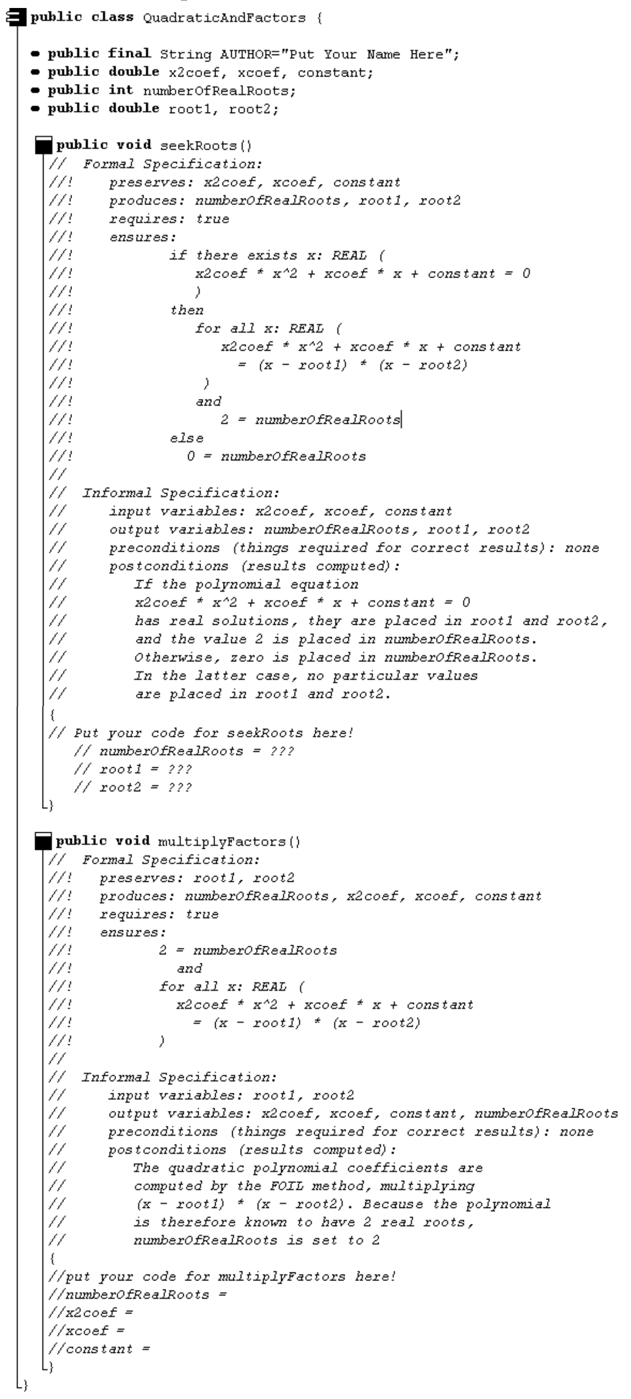


**Figure 2: Example User Interface**

The user interface is shown in Figure 2. The entire application has been designed and implemented in advance, except for the methods invoked when the user clicks on the buttons. One method will compute the roots of a quadratic from its coefficients and the other will compute the inverse relation, multiplying the factors to produce the coefficients. The QuadraticAndFactors.java file provided to the students is shown in Figure 3. All other class files that comprise the application are provided in completed form by the instructor. In addition, students are led through a web-based discussion of the algorithm and data structures required to solve the problem.[3]

In this assignment the student doesn't have to know anything about I/O or parameters, and only needs minimal comprehension of subroutines. It is an advantage pedagogically that in most OO languages, within the implementation of a member method, the components of the distinguished parameter can be referred to without being qualified by 'self' or any other parameter name. This ability affords a simplicity of reference that was the most attractive feature of the otherwise poor practice of accessing global variables. The habits taught in this lab assignment don't have to be unlearned when the student becomes more sophisticated.



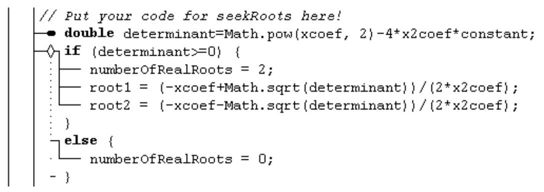
```
public class QuadraticAndFactors {

  public final String AUTHOR="Put Your Name Here";
  public double x2coef, xcoef, constant;
  public int numberOfRealRoots;
  public double root1, root2;

  public void seekRoots()
  //  Formal Specification:
  //!    preserves: x2coef, xcoef, constant
  //!    produces: numberOfRealRoots, root1, root2
  //!    requires: true
  //!    ensures:
  //!            if there exists x: REAL (
  //!               x2coef * x^2 + xcoef * x + constant = 0
  //!               )
  //!            then
  //!               for all x: REAL (
  //!                  x2coef * x^2 + xcoef * x + constant
  //!                    = (x - root1) * (x - root2)
  //!                )
  //!               and
  //!                  2 = numberOfRealRoots
  //!            else
  //!              0 = numberOfRealRoots
  //
  //  Informal Specification:
  //     input variables: x2coef, xcoef, constant
  //     output variables: numberOfRealRoots, root1, root2
  //     preconditions (things required for correct results): none
  //     postconditions (results computed):
  //        If the polynomial equation
  //        x2coef * x^2 + xcoef * x + constant = 0
  //        has real solutions, they are placed in root1 and root2,
  //        and the value 2 is placed in numberOfRealRoots.
  //        Otherwise, zero is placed in numberOfRealRoots.
  //        In the latter case, no particular values
  //        are placed in root1 and root2.
  {
  // Put your code for seekRoots here!
     // numberOfRealRoots = ???
     // root1 = ???
     // root2 = ???
  }

  public void multiplyFactors()
  //  Formal Specification:
  //!    preserves: root1, root2
  //!    produces: numberOfRealRoots, x2coef, xcoef, constant
  //!    requires: true
  //!    ensures:
  //!           2 = numberOfRealRoots
  //!             and
  //!           for all x: REAL (
  //!             x2coef * x^2 + xcoef * x + constant
  //!               = (x - root1) * (x - root2)
  //!           )
  //
  //  Informal Specification:
  //     input variables: root1, root2
  //     output variables: x2coef, xcoef, constant, numberOfRealRoots
  //     preconditions (things required for correct results): none
  //     postconditions (results computed):
  //        The quadratic polynomial coefficients are
  //        computed by the FOIL method, multiplying
  //        (x - root1) * (x - root2). Because the polynomial
  //        is therefore known to have 2 real roots,
  //        numberOfRealRoots is set to 2
  {
  //put your code for multiplyFactors here!
  //numberOfRealRoots =
  //x2coef =
  //xcoef =
  //constant =
  }
}
```

**Figure 3: Example Specification**

The formal specifications provided conform to the stylized approach advocated by the RESOLVE [5, 6] project at Ohio State, consisting of several clauses characterized by the modifiers *preserves*, *produces*, *consumes*, *alters*, *requires*, and *ensures*. These clauses establish a typical

[3] See http://math.otterbein.edu/sigcse/quadratic.htm

contract-type constraint on the behavior of the method with respect to the object state.



```
// Put your code for seekRoots here!
double determinant=Math.pow(xcoef, 2)-4*x2coef*constant;
if (determinant>=0) {
    numberOfRealRoots = 2;
    root1 = (-xcoef+Math.sqrt(determinant))/(2*x2coef);
    root2 = (-xcoef-Math.sqrt(determinant))/(2*x2coef);
}
else {
    numberOfRealRoots = 0;
}
```

**Figure 4: seekRoots Method Solution**

The informal specifications are provided primarily as a tool in teaching the students to read and understand the formal specs. As such, they tend to contain some ambiguity while, hopefully, conveying some intuition to the student.

## 4. Related Efforts

### 4.1 Previous Art

The need for precise specification has been elegantly stated in documents published by the Eiffel project [7]. However, they err pedagogically by assuming that a beginning student should be involved in writing those specifications! Support for the idea that monolithic applications are a bad idea comes from the flurry of textbooks with "Procedures Early." The folly of the procedures early approach was documented in the "Bandwagons" paper [2] (without many recommendations) and in the "Heresy" paper by Pattis [8] recommending Read/Call Early as an alternative.

One of the authors circulated a position statement locally several years ago supporting the implementation of procedure bodies as the correct starting point for CS1. This ultimately came to fruition two years ago with the development of a web-based set of materials using the approach.

Independently, Monash has supported a very similar approach to ours. Because their original software was based on a special language and was unavailable for our platform, we failed to recognize the logical similarity of their approach until recently.

### 4.2 Future Research

We have experimented going one level deeper into the machine by having the student translate the meaning of a program into classical flowchart language. We think this has high potential for increasing student comprehension of sequence, selection, and iteration constructs. Note that we are not advocating *using* flowcharts for program development (which is largely discredited), but rather only translating the meaning of programming language constructs *into* flowcharts. As you nest one construct inside another, it can become quite a good exercise for the student to decipher the meaning as a flowchart. We now support these flowchart exercises a part of the Jarel environment.

Students have a cognitive mismatch when it comes to assignment statements. Because they have just learned algebra, and the symbology is similar, they somehow think that when they type in what appears to be an equation that the computer is going to solve it. One way to overcome this misconception might be to have them translate expressions into several assignment statements, only allowing one operator per statement. What other kinds of exercises might be useful?

We are also experimenting with predictive exercises in the Jarel environment. In this case, the student predicts the next statement to which control will pass, throughout an entire execution of a procedure. If they predict incorrectly, they are shown the actual line to be executed next, and they continue from there. For each run, they are given a score of incorrect and correct predictions. This exercise is at the comprehension level.

## 5. Conclusion

We advocate the application of educational pedagogy to the development of computer science curriculum. Specifically, we support the use of Bloom's Taxonomy to help identify topics, exercises, and assignments for CS1 and CS2. Traditional approaches to CS1 and CS2 are not in congruence with cognitive development theory. We have developed a new pedagogy that we call the inside/out approach, which is tied closely to cognitive development. We have developed web-based materials supporting our approach, both in Delphi Pascal and in Java™, and are making them freely available to the community. The BlueJ project from Monash University significantly extends support for our approach within the Java™ language.

Now that we have brought a venerable model of cognitive development to bear on the structure of the CS1 and CS2 curriculum, the path toward significant enhancement of student outcomes seems obvious. We need to develop more compelling experiences and assignments at the lower levels of cognitive development, and work our way more gradually toward the higher levels. The risk is that we develop materials at the right level, but that teach bad practices (such as monolithic applications). We ask the community to help us fill in more details of how we can effectively build from the lower to higher cognitive levels.

## References

[1] Gronlund, N.E. and Linn, R.L. *Measurement and Evaluation in Teaching*, 6th ed., Macmillan, 1990.

[2] Kay, D.G. Bandwagons Considered Harmful, or The Past as Prologue in Curriculum Change. SIGCSE Bulletin, Volume 28, Number 4 (December 1996), 55-58, 64.

[3] Kölling, M. and Rosenberg, J. Blue – A Language for Teaching Object-Oriented Programming. Proceedings ACM SIGCSE Symposium, 1996, 190-194.

[4] Kölling, M. and Rosenberg, J. An Object-Oriented Program Development Environment for the First Programming Course. Proceedings ACM SIGCSE Symposium, 1996, 83-87.

[5] Long, T.J., Weide, B.W., and Bucci, P. Client View First: An Exodus from Implementation-Biased Teaching. Proceedings ACM SIGCSE Symposium, 1999, 136-140.

[6] Long, T.J., et al. Providing Intellectual Focus to CS1/CS2. Proceedings ACM SIGCSE Symposium, 1998, 252-256.

[7] Meyer, B. Applying "Design by Contract", Computer (IEEE), vol. 25, no. 10, October 1992, 40-51.

[8] Pattis, R.E. The "Procedures Early" Approach in CS 1: A Heresy. SIGCSE Bulletin, Volume 25, Number 1 (March 1993), 122-126.

[9] Pattis, R.E. *Karel the Robot: A Gentle Introduction to the Art of Programming*, 2nd ed., Wiley, 1995.