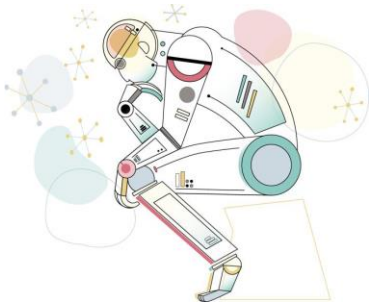
Top Eight Computer Science Education Research Papers



These papers provide a breadth of information about Computer Science Education Research that is generally useful and interesting from a computer science perspective.

Contents

Identifying Student Misconceptions of Programming

Lisa C. Kaczmarczyk
University of California, San Diego
Sixth College, MC0054
lisak@acm.org

Elizabeth R. Petrick
University of California, San Diego
Department of History, MC0104
erpetric@ucsd.edu

J. Philip East
University of Northern Iowa
Department of Computer Science
east@cs.uni.edu

Geoffrey L. Herman
University of Illinois at Urbana-Champaign
Dept. of Electrical and Computer Engineering
glherman@illinois.edu

ABSTRACT
Computing educators are often baffled by the misconceptions that their CS1 students hold. We need to understand these misconceptions more clearly in order to help students form correct conceptions. This paper describes one stage in the development of a concept inventory for Computing Fundamentals: investigation of student misconceptions in a series of core CS1 topics previously identified as both important and difficult. Formal interviews with students revealed four distinct themes, each containing many interesting misconceptions. Three of those misconceptions are detailed in this paper: two misconceptions about memory models, and data assignment when primitives are declared. Individual misconceptions are related, but vary widely, thus providing excellent material to use in the development of the CI. In addition, CS1 instructors are provided immediate usable material for helping their students understand some difficult introductory concepts.

Categories and Subject Descriptors
K.3.2 [**Computers and Education**]: Computer and Information Science Education – *Computer science education*.

General Terms
Human Factors.

Keywords
Curriculum, Concept Inventory, Programming, Misconceptions, Pedagogy, CS1.

1. INTRODUCTION
Most Computer Science Educators will recall times when they were completely baffled by how their students expressed their understanding of a critical topic. Clearly, understanding a student's inaccurate conceptualization is a necessary prerequisite for helping them move toward an accurate conceptualization. Unfortunately we cannot read minds and we cannot speak in depth with every struggling student. Thus, it would be very useful to have a reliable method of rapidly gauging the most important areas of conceptual difficulty, and to reveal in what form these difficulties manifest themselves.

A promising assessment approach is the use of a concept inventory (CI). The original CI was developed by physics educators (Hestenes, et al.) and addressed concepts of Newtonian Force as taught in introductory physics [10]. The authors had previously discovered that many students did not develop correct conceptions of critical topics. In response, the authors produced a multiple-choice examination that could be used by all physics instructors to determine whether their students appropriately understood the concepts of Newtonian Force. Perhaps their most critical contribution has been that instructors can use the inventory results to gain "on the ground" insight into not only the concepts their students are struggling with, but what specific misconceptions they hold. This information can be immediately leveraged to adjust instruction.

Prior to the project of which this paper is part, there was some discussion and preliminary attempts to develop a CI for discrete mathematics [1]. A digital logic CI is currently being developed and is nearly complete [9]. No other CIs have been fully developed for any area of introductory computing.

The results reported here are part of a multi-institutional project to develop concept inventories for three introductory computing topics: digital logic, programming fundamentals, and discrete structures. The process is as follows: previously, Delphi studies were conducted to identify concepts considered both important and a source of difficulty for students [7]. The next step involves interviewing students who have been instructed on each topic to identify their misconceptions. Results for digital logic have been published [8]; initial findings from interviews on programming fundamentals are reported here. As will be discussed in Section 6, these data and additional data currently being collected, will be used to develop, test, and validate the CI instrument.

2. BACKGROUND
Student misconceptions of programming and closely related topics have been studied for some time. Early studies, such as Mayer's work on mental models of the actions of programming statements [16], were followed by Bayman and Mayer examining misconceptions related to individual program statements in

SIGCSE'10, March 10–13, 2010, Milwaukee, Wisconsin, USA.

BASIC [2]. They found that many students had incorrect understandings or outright misconceptions of "much" of even very simple statements. Bonar and Soloway looked more generally at groups of statements to examine student understanding of programming [3]. They recognized that student "step-by-step natural language programming knowledge" interacted negatively with the programming knowledge of formal instruction. These studies touched upon misconceptions, but their primary focus was on larger mental models and theories of cognitive representation, or in some cases discoveries of misconceptions were not fully followed up pedagogically.

Spohrer and Soloway [20] examined the source of programming errors or bugs to see if they result from "misconceptions about the semantics of programming language constructs". They concluded that bugs are more likely to arise from student errors in reading and analyzing specifications and failures to see negative interactions between segments of code. Pea [18] looked beyond language constructs altogether, seeking insight into misconceptions. He found a "superbug"—students' tendency to expect computers to correctly interpret student actions and do the right thing. Confrey [5] examined both theoretical and empirical literature on misconceptions in mathematics, science, and programming and noted that the primary focus of the research was to avoid misconception development though changes in teaching. These studies focus primarily on misconceptions and present important results, however for a wider pedagogical use in CS1, they provide insufficient breadth and depth.

More recent work has often been narrowly focused. Ma, et al. [14] examined the correctness of mental models of assignment (of values and references) that are held by students at the end of a programming course and found a substantial number or erroneous models for even simple assignment. Fluery [6] and Madison and Gifford [15] focused specifically on parameters. Holland, et al. [11] presented misconceptions related to Objects and recommendations for addressing them, however their results were anecdotal and not supported with data.

A variety of other work on misconceptions exists, but from the perspective of providing data needed to develop a CI for CS1, they either replicate the problematic issues above, or else focus on other areas of program related conceptions (correctness and grading) rather than programming conceptions (e.g. Kolikant and Mussai [12] and Sanders and Thomas [19]). Thus there remains a need to investigate student misconceptions across a wide variety of CS1 topics. Conducting in-depth interviews with methodological rigor is one way to provide the broad cognitive understanding needed. In the following sections, we report a first set of results to rectify this situation.

3. INTERVIEW & ANALYSIS PROTOCOL

Eleven students took part in interviews conducted at the University of California, San Diego (UCSD) in spring 2009. Students were recruited from the undergraduate student population who were currently or recently enrolled in Computer Science or Computer Engineering introductory courses CSE8a or CSE11 (two versions of CS1). Participation was voluntary; subjects were recruited through announcements made in courses and via email to CSE lists. Students were compensated for participation in the project.

The primary purpose of the interviews was to reveal misconceptions held by students on an initial group of ten of the thirty-two concepts identified by the Delphi experts [7]. There were eighteen problems, covering the following concepts:

1. Memory Model, References, and Pointers (MMR)
2. Primitive and Reference Type Variables (PVR)
3. Control Flow (CF)
4. Iteration and Loops I (IT1)
5. Types (TYP)
6. Conditionals (COND)
7. Assignment Statements (AS)
8. Arrays I (AR1)
9. Iteration and Loops II (IT2)
10. Operator Precedence (OP)

A secondary purpose of the interviews was to validate the Delphi experts' conclusions that these concepts were indeed difficult.

The interviews were semi-structured and used a modified think-aloud protocol [4]. Choosing a language for code examples was an unavoidable necessity in spite of an overarching goal to develop as language neutral a CI as possible. We selected Java for three reasons. First, Java is currently one of the most widely used introductory programming languages. Second, our Delphi experts explicitly identified a subset of troublesome concepts as Object Oriented (OOP) based. Third, our student population had been taught in Java. It is important to note that not all concepts required that actual code be presented to students in order to reveal misconceptions. A full list of the problems is available from the authors and is expected to be published in a subsequent longer article. In addition, we will address the language dependence issue further in Sections 5 and 6.

There were multiple problems per concept, in order to guarantee that results did not depend on a single question. The majority of problems were covered in at least two distinct variations. Pilot interviews revealed that some concepts were closely related (e.g. Control Flow with other concepts). Thus, misconceptions emerged for some concepts within discussion of problems designed for another concept. Additional interviews and analysis on several of these "overlap" concepts are underway.

Each student was given a subset of the problems. Each interview lasted approximately one hour. With a few exceptions, every student was provided questions for all ten concepts. The exceptions occurred when students worked slowly and time limitations prevented full coverage. To avoid order bias, the problems were given in a semi-random order to each student. The caveat to the randomness of problem ordering is that each student was given one or two simple questions in the beginning to reduce anxiety and acclimate them to the interview process. Students were given the problems on paper, and provided scrap paper to work on if they desired. At no time did the interviewer reveal correct or expected answers to the problems. We collected audio and video recordings of the interviews, along with any written work the students produced. The audio tracks of the interview recordings were transcribed verbatim. Video was used as a back-up and as a visual resource if needed.

We analyzed transcripts and written work from ten of the interviews. Due to equipment failure, one interview was lost. The interviews were analyzed using the following steps of grounded theory and qualitative data analysis as described by Kvale [13], Strauss and Corbin [22], and Miles and Huberman [17]:

1. All of the survey responses were selected for coding in order to avoid bias in selection.
2. All of the survey responses were read and analyzed independently by three researchers: the first and second authors, and a researcher from one of the project's partner institutions (the fourth author).
   a. Each researcher developed codes, operational definitions, and themes grounded in the textual responses.
   b. The three researchers compared their coding and thematic decisions. When there were divergent findings, only those encodings were retained in which all researchers agreed. An inter-rater reliability rating of 96% was achieved.
3. Thirty-two codes with operational definitions were agreed upon. Twenty-five codes addressed the ten targeted concepts.
   a. The codes describe the misconceptions students held and were grouped according to the important and difficult concepts identified by the Delphi experts.
   b. Additional codes addressed other concepts from the full Delphi expert list have been specifically targeted in further interviews during summer and fall 2009.

4. RESULTS

Four themes emerged from the students' misconceptions (see Table 1). Themes 1 and 4 are highly language independent and cover general misunderstandings. Theme 2 involves a number of misconceptions all related to an inability to properly understand the process of while loop functioning. Though not truly language independent, this theme and its misconceptions are applicable across several commonly used contemporary and historic languages. Finally, Theme 3 is a basic lack of understanding of the most fundamental aspects of Object-Oriented Programming.

For the purpose of building a CI, misconceptions are the key data, as they are used to create authentic distracter questions on the instrument. In this paper, we focus on three of the six misconceptions within Theme 1: "Semantics to semantics," (MMR1), "Primitive no default," (PVR1) and "Uninstantiated memory allocation" (MMR4) (see Table 2). Both of the Delphi process concepts these misconceptions fall under (MMR, PVR) were highly ranked overall for importance and difficulty. Of the ten concepts addressed in this set of interviews (see Section 3), these two concepts were ranked highest by the Delphi experts for difficulty.

The first misconception, "Semantics to semantics," (MMR1) occurred when the student inappropriately assumed details about the relationship and operation of code samples, although such information was neither given nor implied. This misconception is language independent although every language will manifest the misconception differently. For example, when examining a list of Java variable definitions and declarations whose inter-relationships are unstated (see Appendix: Problem 1), Student2 explains: "And then have the names of the songs in here, which – but this would be stored in library, I'm assuming, or in the library class. I don't know how they're linked together exactly."
In another example, with a different problem (see Appendix: Problem 2b), Student3 makes incorrect assumptions about connections between variables to the extent that the student makes a mistake concerning the types of the variables. As a result, the student places Objects of different types in an array whose type matches none of them: "And so because there's two arrays, cheese and meats, uh, all those turkey and ham and roast beef are gonna be sorted into the meats array."

In a third example, Student8 completely and repeatedly ignores a variable, because it does not fit with her/his assumptions of how these variables must relate. In a lengthy discussion of the supposed relationships between the variables (see Appendix: Problem 2a), the sole reference (verbally or written) to "sauceType" was the following statement at the very start of the problem discussion: "Usually all the variables go to describing the Object, but I don't think it would describe a sauce."

It should not be surprising that students bring their own assumptions to problems. The Educational Psychology literature has solidly established this basic function of human cognition (e.g. [21]). However, we found it surprising where these assumptions led in terms of confusion between syntax and semantics. Even when an assumption based confusion led to clearly contradictory beliefs and conclusions, the students still could not recognize that their assumptions caused a problem. In one example, a student realized that the syntax did not fit his/her semantic assumptions and, instead of questioning those assumptions, he/she assumed that the syntax must be logically incorrect. Fortunately, this problematic cognitive behavior (for the purposes of learning programming) has also been discussed in the psychological literature and we should be able to draw upon that

Table 1. Themes Emerging From Student Misconceptions

T1: Students misunderstand the relationship between language elements and underlying memory usage.
T2: Students misunderstand the process of while loop operation.
T3: Students lack a basic understanding of the Object concept.
T4: Students cannot trace code linearly.

Table 2. Misconceptions About the Relationship Between Language Elements and Underlying Memory Usage

MMR1	Semantics to semantics	Student applies real-world semantic understanding to variable declarations.
MMR2	All Objects same size	Student thinks all Objects are allocated the same amount of memory regardless of definition and instantiation.
MMR3	Instantiated no memory allocation	Student thinks no memory is allocated for an instantiated Object.
MMR4	Uninstantiated memory allocation	Student thinks memory is allocated for an uninstantiated Object.
MMR5	Off by 1 array construction	Student thinks an array's construction goes from 0 to length, inclusive.
PVR1	Primitive no default	Student thinks primitive types have no default value.
PVR2	Primitives don't have memory	Student thinks primitives without a value have no memory allocated.

field's expertise and resources to customize solutions for computing education.

The second misconception, "Primitive no default," (PVR1) relates to lists of instance variables. This misconception is related to OOP and is a Java specific misconception. Student3 discusses two boolean variables without assigned values (see Appendix: Problem 2b) and states: "I don't think any value is being created for them because there's no assignment there. You know, it's just being declared as a variable." Student5 similarly discusses an integer which is not assigned a value (see Appendix: Problem 2a): "And then int is empty too and it's just creating space to later store an integer."

The third misconception, "Uninstantiated memory allocation," (MMR4) reveals itself when students think that memory is allocated for Objects which have been declared, but not instantiated. This misconception is also related to OOP. For example, Student5 explains how the computer handles memory for the uninstantiated Object "turkey" (see Appendix: Problem 2a): "it's just going to be this blank turkey because we're not setting it to be anything but we're creating like free space to the mater [*sic*] later on declare it."

In another example, involving a similar problem, Student2 discusses the memory allocated for the uninstantiated Object "artist" (see Appendix: Problem 1): "I'm thinking it goes to wherever artist is defined and looks at that class. And I feel like the class would set aside memory.

5. DISCUSSION

We found both unsurprising and surprising results in these interview data. The primary unsurprising, but welcome, result is that the misconceptions we uncovered confirmed the Delphi experts' choice of concepts as difficult for CS1 students.

Two surprising outcomes relate specifically to student misunderstandings. First, the breadth of misconceptions about memory models was unexpected. Memory models are very difficult, but we did not expect such a high number and variety of misconceptions. This finding has an important implication for pedagogy. There are likely to be a diversity of strategies to address memory model misconceptions, without any one quick or universally applicable fix. This challenge is particularly apparent regarding the misconception about students applying semantic assumptions to syntax (MMR1). It will take creative thinking by each instructor, as well as further research, in order to determine the most effective way to leverage these results.

The second surprising outcome relates to Theme 3, not otherwise discussed in this paper: a dearth of even basic conception of an Object. Some students had not formulated misconceptions about Objects, as they had no conceptions at all. During the interviews, they either froze, admitted with some embarrassment to having no idea what an Object was, even when prompted in several ways, or simply changed the subject. This extreme difficulty is being further investigated and results will be reported in a future publication. Meanwhile, one important implication of a lack of knowledge about Objects is that perhaps, within the context of particular student populations, instructors can take a step back and re-think how to introduce the concept of Objects, and focus explicitly on what they consider most critical about Objects in their particular incarnation of CS1.

6. FUTURE WORK

Our findings are representative of our participant population. However, many of the misconceptions we found are generally believed to be universal, but play out differently in different languages, and as such will need to be dealt with in the inventory. As we move forward in developing the inventory, we will further address issues of language dependence. We are currently evaluating options to address this concern. We will also need to address issues of OOP. OOP was an important category of concern to the Delphi experts, and thus must be included. However, we also want to make the inventory as flexible as possible, because at some point in time OOP may no longer be the dominant paradigm. The tension between these competing needs may be our most challenging task.

In following good grounded theory based protocols we have already used the data gathered so far to inform our next steps. First, we have completed a set of interviews conducted in Summer, 2009 that address the remaining Delphi expert concepts as well as the "overlap concepts". We also conducted interviews in the Fall, focusing on concepts which we determined needed additional investigation. Additional interviews are currently taking place at a partner institution to broaden the demographic of student subjects. Next, we will build and test the inventory. Pilot tests will take place at multiple institutions with diverse populations and multiple languages. Many of the original Delphi experts have expressed interest in taking part in initial field tests. Pilot inventory test results will provide feedback about how to improve the inventory questions so that the instrument will be useful to the broadest population and demographic possible.

7. CONCLUSION

We have presented initial results describing three important misconceptions held by CS1 students, along with four broad themes encompassing a larger group of misconceptions. The misconceptions detailed in this paper explore memory model representation and default value assignment of primitive values. These data provide immediately useful information for CS1 instructors to help them understand their students' misconceptions. Finally, these results will be merged with additional data being gathered, and used in the development and validation of a CI for Programming Fundamentals.

8. ACKNOWLEDGMENTS

This work was supported by the National Science Foundation under Grants DUE-0618589, DUE-0618598, DUE-0943318, and CAREER CCR-03047260. The opinions, findings, and conclusions do not necessarily reflect the views of the National Science Foundation or the authors' institutions.

9. APPENDIX

Problem 1. You are setting up a database of information about all the songs you own. Each song has certain information associated with it. Diagram (or use pseudo-code) how this information would be stored in memory:

```
Library library = new Library();
SongList[] songList = new SongList[3];
Genre genre;
Artist artist;
Title title;
Album album;
```

```
int trackNumber = 2;
int year = 1961;
int rating = 5;
```

Problem 2a. You are setting up a database of information about sandwich ingredients. There are a number of information items associated with your database. Diagram (or use pseudo-code) how this information would be stored in memory:

```
Cheese[] cheeses = new Cheese[4];
Meat[] meats = new Meat[2];
Turkey turkey;
Ham ham;
RoastBeef roastBeef;
boolean lettuce = true;
boolean tomato = true;
SauceType sauceType = new SauceType();
int numMeat;
int numCheese;
```

Problem 2b was identical to 2a except for the following two declarations:

```
boolean lettuce;
boolean tomato;
```

10. REFERENCES

[1] Almstrum, V. L., Henderson, P. B., Harvey, V., Heeren, C., Marion, W., Riedesel, C., Soh, L., and Tew, A. E. 2006. Concept inventories in computer science for the topic discrete mathematics. In *ACM SIGCSE Bulletin,* 38, 4 (Dec. 2006), 132-145.

[2] Bayman, P. and Mayer, R. E. 1983. A diagnosis of beginning programmers' misconceptions of BASIC programming statements. *Commun. ACM* 26, 9 (Sep. 1983), 677-679.

[3] Bonar, J. and Soloway, E. 1985. Preprogramming knowledge: a major source of misconceptions in novice programmers. *Hum.-Comput. Interact.* 1, 2 (Jun. 1985), 133-161.

[4] Bowen, C. W. 1994. Think-Aloud Methods in Chemistry Education. In *Journal of Chemical Education*. 71, 3 (Mar. 1994), 184-190.

[5] Confrey, J. 1990. A review of the research on student conceptions in mathematics, science, and programming. *Review of Research in Education,* 16, 3 (1990), 3-56.

[6] Fleury, A. E. 1991. Parameter passing: the rules the students construct. In *Proceedings of the Twenty-Second SIGCSE Technical Symposium on Computer Science Education* (San Antonio, Texas, United States, March 07 - 08, 1991).

[7] Goldman, K., Gross, P., Heeren, C., Herman, G., Kaczmarczyk, L., Loui, M. C. and Zilles, C. 2008. Identifying important and difficult concepts in introductory computing courses using a Delphi process. In *Proceedings of the Thirty-Ninth SIGCSE Technical Symposium on Computer Science Education* (Portland, OR, United States, March 12-15, 2008).

[8] Herman, G. L., Kaczmarczyk, L., Loui, M. C., and Zilles, C. 2008. Proof by incomplete enumeration and other logical misconceptions. In *Proceedings of the Fourth International Workshop on Computing Education Research* (Sydney, Australia, Sep. 06 - 07, 2008).

[9] Herman, G. L., Loui, M. C., and Zilles, C., Creating the Digital Logic Concept Inventory. In *Proceedings of the Forty-First ACM Technical Symposium on Computer Science Education*, Milwaukee, WI, March 10-13, 2010.

[10] Hestenes, D., Wells, M., and Swackhamer, G. 1992. Force concept inventory. *The Physics Teacher,* 30 (Mar. 1992), 141-158.

[11] Holland, S., Griffiths, R., and Woodman, M. 1997. Avoiding Object misconceptions. In *Proceedings of the Twenty-Eighth SIGCSE Technical Symposium on Computer Science Education* (San Jose, California, United States, February 27 - March 01, 1997).

[12] Kolikant, Y. B-D. and Mussai, M. 2008. "So my program doesn't run!" Definition, origins, and practical expressions of students' (mis)conceptions of correctness, *Computer Science Education*, 18, 2 (Jun. 2008), 135-151.

[13] Kvale, S. 1996. *Interviews: An Introduction to Qualitative Research Inquiry*. Sage Publications, Thousand Oaks, CA.

[14] Ma, L., Ferguson, J., Roper, M., and Wood, M. 2007. Investigating the viability of mental models held by novice programmers. In *Proceedings of the Thirty-Eighth SIGCSE Technical Symposium on Computer Science Education* (Covington, Kentucky, United States, March 07 - 11, 2007). SIGCSE '07.

[15] Madison, A. and Gifford, J. 2003. Modular programming: Novice misconceptions. *Journal of Research on Technology in Education*, 34, 3 (Spr. 2003), 217-229.

[16] Mayer, R. E. 1981. The Psychology of How Novices Learn Computer Programming. *ACM Comput. Surv.* 13, 1 (Mar. 1981), 121-141.

[17] Miles, M.B. and Huberman, A.M. 1994. *Qualitative Data Analysis: An Expanded Sourcebook, 2nd Edition.* Sage Publications, Thousand Oaks, CA.

[18] Pea, R. D. 1986. Language-independent conceptual "bugs" in novice programming. *Journal of Educational Computing Research*, 2, 1 (1986), 25-36.

[19] Sanders, K. and Thomas, L. 2007. Checklists for grading Object-oriented CS1 programs: concepts and misconceptions. In *Proceedings of the Twelfth Annual Conference on Innovation and Technology in Computer Science Education* (Dundee, Scotland, June 25 - 27, 2007).

[20] Spohrer, J. C. and Soloway, E. 1986. Alternatives to construct-based programming misconceptions. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems* (Boston, MA, United States, April 13 - 17, 1986).

[21] Stanovich, K. E. 2003. The Fundamental Computational Biases of Human Cognition: Heuristics That (Sometimes) Impair Decision Making and Problem Solving. In *The Psychology of Problem Solving*, J. E. Davidson and R. J. Sternberg, Eds. Cambridge University Press, Cambridge, UK, 291-342.

[22] Strauss, A. and Corbin, J. 1998. *Basics of Qualitative Research*. Sage Publications, Thousand Oaks, CA

Undergraduate Women in Computer Science: Experience, Motivation and Culture

Allan Fisher, Jane Margolis and Faye Miller
School of Computer Science, Carnegie Mellon University
5000 Forbes Avenue, Pittsburgh, Pennsylvania 15213

Abstract

For the past year, we have been studying the experiences of undergraduate women studying computer science at Carnegie Mellon University, with a specific eye toward understanding the influences and processes whereby they attach themselves to or detach themselves from the field. This report, midway through the two-year project, recaps the goals and methods of the study, reports on our progress and preliminary conclusions, and sketches our plans for the final year and the future beyond this particular project.

1. Background

The goal of our project has been to understand women's attachment and detachment from computer science, and to find ways for CMU to intervene at the undergraduate level in favor of gender equity in computer science. Women are underrepresented in computer science at CMU and in other higher education institutions across the nation: for example, they receive 18% of the bachelor's degrees in CS at the top 12 research departments [1]. Since computers and information technology play an increasingly pervasive role in education and careers, this underrepresentation is critical, not only for the women whose potential may go unrealized, but also for a society increasing dependent on the technology.

Clearly part of the low representation of women in CS at the undergraduate level is inherited from the secondary school level, where girls do not participate in computer science courses and related activities as much as boys [7]. There is a gap between male and female enrollment in high school computer science courses that increases as students progress from introductory to more advanced CS courses [8]. Females have been only about 12% of AP computer science AB exam takers over the past five years (College Board, private communication). As we learn more about the different ways that students attach to and detach from computer science, we will apply the lessons learned to the design of pedagogical, administrative, and social methods aimed at both attracting and retaining women students.

This paper reports our findings in the initial phase of our research. This part of the research is based on gathering students' accounts of their histories and thoughts about computer science. We have been studying students' perceptions of attachment and detachment from the discipline. In order to conceive of the most effective interventions, we are working to understand the relative importance of the factors that have the greatest bearing on the low numbers of women in the field.

2. Ethnographic Methodology

We have been using ethnographic methods [4,5], with interviews being the primary source of our data. We regard the students as expert witnesses in their own world, and try to ask the questions that will enable them to best elucidate their thoughts about computer science. It is then up to us to note significant themes and patterns. We are not testing hypotheses, but rather are generating testable hypotheses about students' attachment and detachment.

Participants

The participants of our study are:

1. CMU Computer Science male (29) and female (20) students (first-year to senior);
2. Two selected samples of female non-CS majors: 9 students doing well (receiving an A at midterm) in a non-majors' programming class.

Analyzing the Data

Every interview is tape recorded. The interviews are transcribed and the transcripts are entered into HyperResearch, a commercial computer program developed to assist in qualitative data analysis. After coding the interviews for events and themes, the coder writes what we call a "narrative summary." This is our attempt to keep the participants story as whole as possible, to avoid "context stripping." We have worked very hard negotiating the tension between presenting our data as full portraits and the almost necessary "fracturing" of the data into discrete elements so that we can detect patterns across groups and categories (see [4, p. 63]).

Reliability

We are aware of the risk of compromised data analysis and we are continually asking ourselves how can we get the most accurate and detailed picture of the situation.

We have three main defenses against drawing biased or unwarranted conclusions. First, we are refining the coding scheme to a fine level of detail, which tends to decrease the subjectivity of the classification of elements of students' accounts. Second, the cross-disciplinary makeup of our research team helps to expose implicit preconceptions. Finally, we will be holding regular focus groups this year to continually return to the participants, and other groups of CS students, to double-check what we are hearing and hypothesizing.

3. Initial Findings

In this section we briefly discuss our "working hypotheses" from the first year of interviews.

Gender Gap in Previous Experience

During the interviews with first-year CS students, many of the women speak of feeling less prepared than the other students in the department. To obtain more insight into this issue, we distributed a survey questionnaire to all first-year CS students regarding their experience and knowledge of computers prior to attending CMU. Our study confirms a significant gap between male and female prior experience, noted in other studies as well [2,3]. It is notable that 40% of the male respondents from the CMU first-year class passed the AP exam, thereby placing out of the CMU introductory level computing class. None of the first-year women placed out. Also, we found a correlation between females students' sense of feeling less prepared and their actual experience with computers prior to CMU.

Gap Between Perceived and Actual Ability

Despite this difference in how students evaluate themselves, there is a gap between women's perceived ability and their actual performance. Despite their modest estimates of their own standing in the class, three out of the seven first-year students made the Dean's List (which turned out to be about the top third of the class) in the first semester, and six of the seven women made a B or A average for the first year.

Hacking Not a Prerequisite for Success

Many of the female students have entered the department with very little computer experience, yet they do well. Their stories counter the suggestion that prior computing experience is necessary to do well in undergraduate computer science. Their stories of success raise some challenges to widely-held beliefs of who does computer science. Their success is not without costs, though -- they often go through a very difficult period of adjustment, facing tremendous self-doubt and feelings of isolation and inadequacy. Nonetheless, it is clear that one need not have been a high school hacker to major in CS. Our findings have become an important talking point for prospective students, and may have contributed to the improved recruitment of women students for the coming year.

Confidence Gap Narrows

Based on the gender gap in previous computing experience, it is not surprising to find a difference in the confidence levels of male and female first-year students. Female first year students report themselves as being significantly lower in computing experience, preparedness for their computer science courses, and ability to master the course material than the males. In contrast, in response to a first semester survey, the males' stated confidence is quite high. For example, 53% of the men rated themselves as highly prepared for their classes, whereas none of the women rated themselves as highly prepared. 50% of the men reported themselves as having an expert level of at least one programming language prior to CMU, whereas none of the women reported themselves as having an expert level of knowledge of a language. We have heard in the interviews how this gender gap in confidence affects the women students' experiences in the program. In our first-year interviews female students commonly refer to how much more other students (males) know, and question whether they belong.

What we were surprised to hear from the upperclass women was that confidence seems to rise, rather than fall, as women progress through their junior and senior years. This is contrary to the findings of studies from other disciplines. Junior and senior women talked to us about a leveling process, which occurs as the course material gets more difficult for everyone by the junior year, and as women's hard work and discipline has paid off. We asked first-year students and upperclass students to rate their feeling of preparedness for their CS classes compared to classmates, and their ability to master the course material, for their first semester and their current semester. For both groups, those students who felt least prepared at the beginning experienced the greatest increase in feelings of preparedness over time. Women rate themselves lowest in initial feelings of preparedness, and show the most increase (1.1 rise in preparedness for first-year women on a scale of 1-5, versus a .3 rise for men.)

If we continue to hear this, as we interview more students, this finding could be very important for increasing women's confidence about themselves in this field.

Attachment Begins at Home

Research on women in the sciences has highlighted the importance of family influence on students' exposure to and interest in majoring in the sciences [9]. Our interviews certainly confirm this. Most of the students, male and female, were first introduced to computing by a parent who either works on computers themselves or brings one home for the child. School is almost incidental, except in a few cases. Male students, with only a few exceptions, reported owning their own computer, or having the family computer in their room, by an early age. Only one of the seven first-year women reported having her own computer prior to CMU.

While females are also influenced by a parent at home, we hear a difference between the females and males that we believe to be important and deserving of further inquiry. Females' stories are filled with descriptions of *watching* their dad work at the computer, or having their older brother *show* them how he programs the machine. From there, their interest is sparked, and some do become active in computing activities in high school, but their participation is much more qualified than the males'. There seems to be less tinkering, less unguided exploration and less obsession. Indeed, even the female who was president of her high school computer club, says in reference to computing, "I never really got totally into it."

Males: Computers as the Ultimate Toy

Several males describe epiphany moments from their earliest (before 10) computing experiences, sometimes receiving the sense that this is what they wanted to do for the rest of their lives. They become consumed early on and their computer activities become a consistent part of their identity. One student answers the question "Can you tell me how you got interested in computers?" this way:

Well, I think it was sometime in middle school, sixth grade about then, my dad borrowed a computer from a friend, it was an old black and white Macintosh, just totally self contained one unit thing, and I remember just playing with that all the time and trying to figure stuff on it. And that got me really hooked ... I was really getting into figuring things out on computers and I just knew that that was going to be something for me.

Other male students respond likewise:

I started playing around with computers before I canremember...I think I supposedly knew how to type on a machine before I could write....

I liked to play games a lot when I was growing up on them. They just seemed to be really integral to how I like to express my creativity....

But I like just what a computer can do. I don't know why it interests me so much...They say kids like to take things apart and see what makes them go and I do that a lot....

My mother brought me a computer back in Alabama when I was four years old and I guess ever since it has been me playing video games, thinking "WOW, how did they do that"?

The male narratives are filled with descriptions of the*computer itself as an alluring object*. The computer is the ultimate toy and they get "hooked."

Females: Computing with a Purpose

The female stories have a very different sound: When the first-year females talk about their personal history with computers, their narratives are not filled with long and detailed accounts of all the different activities they have done at the computer. They do not describe years of unguided exploration. They contextualize their interest in computer science, instead, within a larger purpose: what *they can do in the world*. One female student who talks about her "lust" for technology, continues to explain that she is "not interested in the nitty-gritty of computers", but sees herself as "exploiting" the department --- getting all the computer knowledge she can, to then be able to apply it to puppetry and art. The women we are interviewing describe computers as a *tool to use within a broader context* of education, medicine, communication, art and music.

What I would really like to do is teach...would like to minor in education and how computers affect education and what is the most efficient way to use them in education.

I really wanted to get people together...how can this change the world as we see it today. You can get people together. You can provide information.

I think with all this newest technology there is so much we can do with it to connect it with the science field, and that's kind of what I want to do(study diseases). Like use all this technology and use it to solve the problems of science we have, the mysteries."

You tend to think of computer scientists aspeople that sit in front of computers all day...doing netscaping, that sort of thing. I can't stand doing that. I have to be actually making something, something productive, or I get depressed.

This is not to say that women totally lack interest in the computing process itself. Female students describe computing as enjoyable, interesting and "hard but fun." Two of the women who had previous work experience in computing lab environments describe the experience as "awesome." But, most of the women talk more about the uses of computing. We have also heard older males, as they progress in the program, articulate more interest in the larger context of computing.

Computer Science: An Acquired Taste

Rather than epiphany moments as described by the males, females stories seem to reflect a process over time, in which their interest in computers evolves. Due to the variety of obstacles girls/women find in their computing path, it may take women more time to be drawn to computers (Sheila Tobias, personal communication). Developing an interest over time was expressed by one of the first year female students:

My dad's always been into computers... We always had a computer in the house. It's always been like, we always like tinker around with them, play games, stuff like that. I never really got totally, like totally into it. I never started programming. But, I don't know, I just kind of found that I really enjoyed working with computers over time... So now I am here and I get it more than I would have. And I'm pretty good at like fooling around with something and just kind of getting it to work, I guess you can say.

Similarly, an international woman senior student, who had no computing experience at all prior to coming to CMU, described her experience with computing as "an acquired taste." As she progressed in the program she became more comfortable in the department and with the course work and actually developed a new-found interest in the field. This certainly speaks *against* the notion that women are cognitively ill-equipped to do CS. Rather, it bolsters the notion of cultural artifacts that stand in between women and computing.

Decision to Major in CS: Love and Pragmatism

Reasons for becoming interested in computer science and selecting it as a major differ among the men, American women, and international women in our sample. We asked the students both why they became involved in computing, and why they chose CS as a major; the most salient reasons cited are plotted below as percentage citing a reason for majoring vs. percentage citing it as a reason for attachment.

As Figure 1 shows, all of the men interviewed cited an intrinsic interest in computers and computing as a reason for becoming involved in the field. While they cited a number of other factors (notably games, classes and the influence of peers) for their initial attachment, interest alone was the primary driver of their decisions to major in CS.

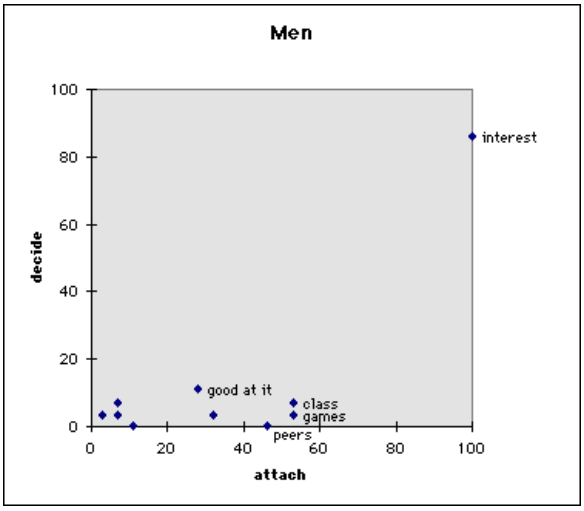


Figure 1: Majoring vs. Attachment (Men)

American women, while also citing intrinsic interest as a motivator, rank class experiences and their sense of the promise of the field and its future high among reasons for majoring. It is interesting to note that while they reported encouragement from family and teachers as reasons for attachment, these do not rank high in terms of reasons for majoring. Also notable is that few cite games or peer interactions as reasons for attachment.

Perhaps the most interesting finding in our interviews concerns the international women. Among this group, pragmatic factors (employability, the image of CS as a pragmatic choice among math, science and engineering-related fields) dominate both attachment and choice of major. While all of the US students cited interest as a reason for attachment, fewer than 60% of the international students did so. This stands in sharp contrast to Seymour's findings that interest above any other factor is critically important in retaining women in the sciences [9]. Whether this contrast is due to cultural differences and/or to the circumstances under which international women find themselves studying in the US bears closer study.

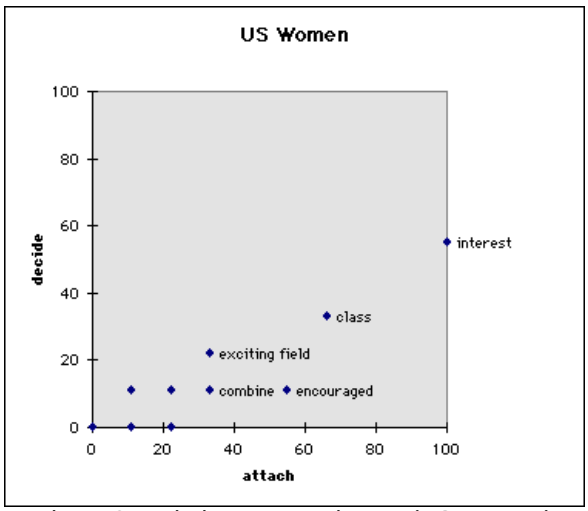


Figure 2: Majoring vs. Attachment (US Women)

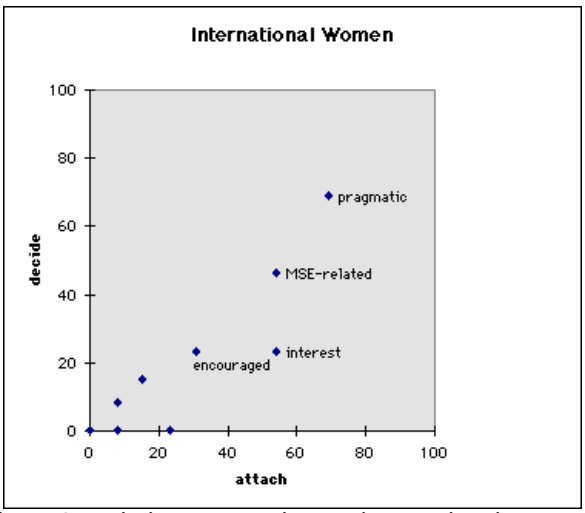


Figure 3: Majoring vs. Attachment (International Women)

Perceptions of the Field

A large fraction of the CS experience in the first year is programming. Upper class students comment on how they realized in their Junior and Senior years that Computer Science is more than programming, and they often express relief at that. First- year students who have had the benefit of hearing from upper class students, and who have regular contact with faculty first-year advisors, also seem to know that programming is not the be-all and end-all. But, outside of the School of CS, we hear students' beliefs that computer science *is* programming.

Students from the Information Science major, who share much interest in computers and computing, state their disinterest in Computer Science largely based on the emphasis on programming in the CS curriculum. Women students whom we interviewed in the non-major introductory programming course (from a variety of fields) describe their fear, dislike, intense anxiety, disinterest in programming when they began the class. Most of these students express an awakening in the course to the fact that programming can actually be interesting and satisfying to understand. But most are not motivated to continue to a deeper level, and they associate the CS major with programming.

Geek Mythology: Lore about Being in CS

Interviews with all students are filled with local lore and impressions about CS and about the CMU department in particular. The beliefs we hear over and over again are that:

- computer science students have a single-minded focus and talk incessantly about computing
- CS is the department with the really smart students
- the work load is extremely heavy (with special emphasis on the amount of time that it takes to complete programming assignments)

The stereotype is clearly the myopic, narrowly focused, young male who sits at his computer all day. This is how one of the female CS students describes this type of student and how they affect her:

I ask them, "How can you sit in front of a computer for eight straight hours and then when you go home you start to play on computer games again?" And then they say, "oh, because it's fun." I say, "don't you spend time with your friends?", and they say, "no, I just like sitting in my room and just play these games." So I just felt really different because, I don't know, I don't know... if you want to major in computer science, what you are supposed to do? Like just play on the computer all day? I don't, so I felt different.

It is important to note that most of the CS students (both male and female) we interviewed feel they do not match the stereotype: their interests are varied (including sports, theater, poetry) and not isolated to computer science. The gap between reality and stereotype of the qualities needed to be a successful CS major and who CS majors are is important to analyze, because the stereotypes work against gender equity. If we can dispel the perceptions of most CS students being immature males who burrow into their computers for all forms of satisfaction, there is hope for progress.

Climate Issues

From our interviews we hear a tension between some women who believe gender to be a non-issue, and other women who feel disrespected in the department because of their gender. The former group feel experienced at handling male environments, feel at ease, and believe attention to gender is unnecessary. The latter group of women describe concerns and/or unhappiness about the male environment and/or the way they are treated. For instance, one first-year woman describes unwanted romantic attention when she is trying to complete her assignments in the computer lab; another describes her alienation from the culture of CS, which she attributes to testosterone run amok.

Peers

It is not unusual for a woman student, within one semester, to report differing impressions: that most of her male peers are willing to help, and that male students make her feel so stupid when she asks them a question. Several of the women talk about the male students knowing so much more than they do.

We asked every student for their views on why there are so few women in computer science. As we understand their comments at this point, we have found some of the male interviews to be particularly provocative. Many of them have concluded, from their school and family experience, that women just aren't interested in the subject the way males are. Most of the males describe school classes with only a very few women, and families where mothers are "unable to plug in the machine" etc. One male student added that he doesn't think he has had a computer conversation with a girl in his life. We wonder how this socializing history may influence male students' attitudes towards women students and faculty in the program.

Faculty and Teaching

While one upperclass student who had transferred out of the department reported negative experiences with an unsupportive and unhelpful professor, most of the female students either have felt supported by the faculty, or have not voiced any complaints. It is not clear to us whether the disparity between this finding and the commonplace occurrence of behavior discouraging to female students in other studies is due to a favorable environment at CMU, failure of the students to notice those behaviors, or the peculiar effects of especially low ratios of women in classes. We will need to carry out more classroom observations and focus group discussions to clarify this point.

4. Conclusions and Next Steps

As we work forward from these observations toward a program of interventions, the three sets of issues we will be working to elucidate are those surrounding individual and cultural conceptions of computer science, those involving pedagogy, and those involving institutional culture. In all cases, we will be working to sort the essential features of computer science from the

accidental (and perhaps harmful), and to understand how perceptions and misperceptions are formed and influence students' decisions. We will be asking how we can improve both the reality of the computer science program and its culture, and the accuracy with which they are perceived by computer science students, other students and prospective students.

A key question that pervades students' accounts of their relationships with computing is their understanding of the nature of the field, in both its intellectual and social aspects. Considering that a wide range of conceptions of computer science exists among faculty, what about the nature of the field gets translated to existing and potential female and male students? Among the issues that seem to deter women from pursuing computer science is the conception that it is narrowly focused on programming and other technical issues, and that people who enter CS are forced (or choose) to be narrowly focused themselves. Even students within CS carry this stereotype of others, while denying it applies to them. In our ongoing study, we will work to elucidate these issues, and to develop ways of communicating the "big picture" earlier and more accurately to first-year and prospective students.

Part of this effort will be to sharpen our picture of the CS education process and ways in which it could be improved. If women prefer to learn about the computer in a purposeful context (i.e. "programming for a purpose, not just to program"), does the curriculum respond? Are assignments more in line with what seems to be young male desires, such as focusing predominantly on the machine? Although the department has made improvements, it is arguably still true that the early curriculum (here and nationwide) fails to paint a complete picture of the field's possibilities [3,6]. We are also aware of the possibilities of different pedagogical approaches to programming [10]. One question we are analyzing is whether females and males differ in their cognitive preferences in programming.

Another issue we plan to address is the prevailing conception of gender in CS among the student body. The only significant "chilly climate" issue raised in our interviews concerns the attitudes of fellow students. This is a delicate issue, posing substantial risk of backlash against clumsy consciousness-raising efforts. In seeking effective means of shifting the prevailing culture, we will be asking students about the roots of their assumptions about women and computer science, and about experiences that have changed or might change them.

Acknowledgment

We gratefully acknowledge the support of the Alfred P. Sloan Foundation.

Bibliography

1. Andrews, Gregory R. (1996). "1995 CRA Taulbee Survey: New enrollment in Ph.D. programs drops," Computing Research News, March, pp. 6-9.
2. Kersteen, Z., Linn, M., Clancy, M., & Hardyck, C. (1988). "Previous Experience and the Learning of Computer Programming: The Computer Helps Those Who Help Themselves." Journal of Educational Computing Research, 4(3): 321-333.
3. Martin, C. Dianne, ed. (1992). In Search of Gender-Free Paradigms for Computer Science Education. ISTE. Eugene, OR.
4. Maxwell, Joseph A. (1996). Qualitative Research Design: An Interactive Approach. Sage Publications, Thousand Oaks, CA.
5. Miles, Matthew, and Huberman, Michael. (1994). Qualitative Data Analysis, 2nd edition. Sage Publications, Thousand Oaks, CA.
6. Rosser, Sue V. (1995). Teaching the Majority. Teachers College Press, New York.
7. Sanders, Jo. (1995). "Girls and Technology: Villain Wanted". in Teaching the Majority, Sue V. Rosser, ed., Teachers College Press, pp. 147-159.
8. Schofield, Janet Ward. (1995). Computers and Classroom Culture. Cambridge University Press, New York.
9. Seymour, Elaine and Nancy M. Hewitt. (1994) Talking About Leaving. Factors Contributing to High Attrition Rates Among Science, Mathematics & Engineering Undergraduate Majors: Final Report to the Alfred P. Sloan Foundation on an Ethnographic Inquiry at Seven Institutions. University of Colorado. Boulder.
10. Turkle, Sherry and Seymour Papert. (1990). "Epistemological Pluralism: Styles and Voices within the Computer Culture." in Signs: Journal of Women in Culture and Society. 16(1), 128-157.

Improving the CS1 Experience with Pair Programming

Nachiappan Nagappan[1], Laurie Williams[1], Miriam Ferzli[2], Eric Wiebe[2,] Kai Yang[1], Carol Miller[1], Suzanne Balik[1]

[1]Department of Computer Science
[2]Department of Math, Science and Technology Education
North Carolina State University, Raleigh, NC 27695
{nnagapp, lawilli3, mgferzli, wiebe, kyang, miller, spbalik}@unity.ncsu.edu

Abstract

Pair programming is a practice in which two programmers work collaboratively at one computer, on the same design, algorithm, or code. Prior research indicates that pair programmers produce higher quality code in essentially half the time taken by solo programmers. An experiment was run to assess the efficacy of pair programming in an introductory Computer Science course. Student pair programmers were more self-sufficient, generally perform better on projects and exams, and were more likely to complete the class with a grade of C or better than their solo counterparts. Results indicate that pair programming creates a laboratory environment conducive to more advanced, active learning than traditional labs; students and lab instructors report labs to be more productive and less frustrating.

Categories & Subject Descriptors

K.3 [**Computers & Education**]: Computer & information science Education- *Computer Science Education*.

General Terms

Management, Human Factors

Keywords

Pair programming, collaborative environment, Computer Science education.

1 Introduction

In industry, software developers generally spend 30% of their time working alone, 50% of their time working with one other person, and 20% of their time working with two or more people. [3] However, most often in an academic environment, programmers must learn to program alone, and collaboration is considered cheating. Unfortunately, this time spent working alone is inconsistent with a student's future professional life in which collaboration is both encouraged and required. In addition, studies show that cooperative and collaborative pedagogies are beneficial for students [6, 7].

In pair programming one person, called the *driver*, is responsible for typing at the computer or documenting a design. The other partner, called the *navigator*, observes the work of the driver, looking for defects in the work of the driver and is an ever-ready brainstorming partner. Research results [2, 8, 11] indicate that pair programmers produce higher quality code in about half the time when compared with solo programmers. These research results are based on experiments held at the University of Utah in a senior-level Software Engineering course. The focus of that research was the affordability of the practice of pair programming and the ability of the practice to yield higher quality code. However, the researchers observed educational benefits for the student pair programmers. These benefits included superior results on graded assignments, increased satisfaction/reduced frustration from the students, increased confidence from the students on their project results, and reduced workload of the teaching staff.

These observations inspired further research directed at the use of pair programming in educating Computer Science students. Educators at the University of California-Santa Cruz [1, 5] and North Carolina State University [9, 10] have reported on the use of pair programming in introductory undergraduate programming courses. Experiments specifically designed to assess the efficacy of pair programming in an introductory Computer Science classroom found that pair programming improved retention rates and performance on programming assignments.

This paper details the results of our experiment carried out at North Carolina State University. We provide results from a larger sample size than previously reported. The remainder of this paper is organized as follows: Section 2 provides a description of the experiment; Section 3 discusses qualitative findings on pair programming in the CS1 laboratory; Section 4 shares the results of our quantitative findings; Section 5 highlights a few challenges we faced during this experiment and Section 6, summarizes our findings and discusses our future work.

2 Experiment

In the 2001-2 academic year, an experiment was conducted in the CS1 course at North Carolina State University. The course was taught with two 50-minute-lectures and one three-hour lab each week. Students attended labs in groups of 24

SIGCSE'03, February 19-23, 2003, Reno, Nevada, USA.

with others in their own lecture section. The lab period was run as a closed lab where students were given a weekly assignment to complete during the allotted time. Lab assignments are "completion" assignments whereby students fill in the body of methods in a skeleton of the program prepared by the instructor. Student grades are based on two midterm exams, one final exam, lab assignments, and programming projects that are completed outside of the closed lab. The programming projects are generative, that is, the students start the project from scratch without any structure imposed by the instructor. The course is a service course and is therefore taken by many students throughout the university. Most students are from the College of Engineering and are either freshmen or sophomores. However, students of all undergraduate and graduate levels may take the course.

The Fall 2001 experiment was run in two sections of the course; the same instructor taught both sections. Additionally, the midterm exams and the final exam were identical in both sections. One section had traditional, solo programming labs. In the other section, students were required to complete their lab assignments utilizing the pair programming practice. When students enrolled for the class, they had no knowledge of the experiment or if their section would have paired or solo labs. In the pair programming labs, students were randomly assigned partners based on a web-based computer program; pair assignments were not based on student preferences. Students worked with the same partner for two to three weeks. If a student's partner did not show up for a particular lab, after 10 minutes, the student was assigned to another partner. If there were an odd number of students, three students worked together; no one worked alone. Closed labs are excellent for controlled use of pair programming [1]. The instructor or teaching assistant can ensure that people are, indeed, working in pairs at one computer. He or she can also monitor that the roles of driver and navigator are rotated periodically.

Our course also includes programming projects that require work outside of the closed lab. We gave the students in both sections the option of working alone or in pairs for these projects. Only students who attained a score of 70% or better on the exams could opt to pair. (We felt those who did not attain a score of 70% or above should not work with a pair on the project lest they rely too heavily on their partner to produce the project.) Most students, who were eligible to pair, chose to pair program on projects. However, the instructors now feel that the 70% eligibility might be unfair to the students, and this practice has been discontinued as of Fall 2002.

Using this Fall 2001 research design, we also completed a study on a larger scale in the Spring 2002 semester. In the fall, 112 students were in the solo section and 87 were in the paired section, whereas in the spring 156 students worked solo and 346 students worked in pairs. Our study was specifically aimed at the effects of pair programming on beginning students. Therefore, we analyzed the results of the freshman and sophomores only. We also only analyzed students who took the course for a grade, concluding that students who audited the class or took it for credit only were not as motivated to excel as other students. This reduced our sample size to N=69 in the solo section and N=44 in the paired section for the Fall semester, and N=102 for the solo section and N=280 in the paired section for the Spring semester.

In our experiment (spanning both Fall and Spring semesters), we examined the following five hypotheses:

H1. A higher percentage of students who have participated in pair programming in CS1 will succeed in completing the class with a grade of C or better when compared with students who have worked solo in CS1.

H2. Students' participation in pair-programming in CS1 will lead to better performance (higher scores) on the examinations when compared with students who have worked solo in CS1. (Examinations are completed solo by all students)

H3. Students' participation in pair-programming in CS1 will lead to better performance on course projects (higher project scores) in that class when compared with students who have worked solo in CS1.

H4. Students' participation in pair-programming will lead to a reduced workload in terms of grading, questions answered, and teaching effort for the course staff when compared with the teaching staff for students who worked solo in CS1.

H5. Students in paired labs will have a positive attitude towards collaborative programming settings when compared with students who have worked solo in CS1.

3 Qualitative Results

Each semester, we observed and codified many paired and solo lab sections. In addition, two focus groups were held, one with a randomly selected group of students and the other with a randomly selected group of lab instructors (LIs). (See focus group technical report [4].) Analysis of qualitative data from lab observations and focus groups strongly support pair programming in the CS1 laboratory. The next sections detail student and lab instructor perspectives on pair programming.

3.1 Students

Solo lab sessions were quiet and appeared to be very frustrating for the students. Frequently, a student needed to wait 10-30 minutes to ask a question, often a fairly simple one. During this waiting period or "down time", students were often very unproductive (i.e. "stuck"). Alternately, paired labs were vocal and interactive. Students in paired labs engaged in extensive discussion throughout the entire lab session, and students seemed to help each other resolve questions. Most often, each pair could piece together the knowledge they needed to figure out questions and remain productive. Because most pairs were self-sufficient, lab instructors had time to get around to more students than in the unpaired sections. Paired students who needed help, found it easy to get help from the LI, and had little "down time." [9]

During the focus group discussion, students stressed the advantages of pairing. Primarily, students brought up the benefits of having their questions answered immediately by their partner rather than having to wait for an LI. Having

someone there while working on problems also seemed to help them pick up on minor errors and to focus on understanding conceptual knowledge.

Since communication skills and collaboration are important components of paired learning, students recognized that the paired labs made them work on these skills. Students realized that the paired format mimics real world settings where people are often randomly matched to work and collaborate on programming projects.

3.2 Lab Instructors

In solo lab sections, the LIs were often overwhelmed with questions. LIs often spent a minimum of five minutes and a maximum of 20 minutes with each student. LIs remained busy answering basic questions for the duration of the lab sessions. In paired labs, instructors spent more time discussing advanced issues with students, rather than answering basic questions.[9] For example, students in paired labs would ask the LIs how to improve their algorithm, or how to apply it to another scenario. Questions from students in solo labs were mostly about fixing syntax errors or getting compilation errors clarified

In the focus groups, the LIs all agreed that implementing the paired protocol gave them flexibility and time to give students equal opportunities for questions, discussions, and other support. As a result of having more time for meaningful exchanges with students, LIs found their jobs more satisfying and rewarding when teaching in paired labs. An added benefit is that LIs of paired labs graded half the number of projects and labs as compared to the LIs of solo labs.

LIs noted that students in paired labs displayed more active participation in their learning than students in the unpaired labs. Paired student questions displayed higher order thinking such as application, synthesis, and evaluation. LIs observed that paired students' efforts and willingness to learn seemed to surpass their "traditional" counterparts.

(H4) We hypothesized that students' participation in pair programming will lead to a reduced workload for course staff. Our qualitative findings support this claim.

3.3 Common Concern

In both focus groups, the students and LIs noted the importance of having "compatible" partners. Two suggestions for constructing compatible pairings were to have them be based on personality type and/or on skill level. We address our research plans in this area in Section 5.

4 Quantitative Findings

In the prior section, we shared our qualitative findings that pairing creates a laboratory environment conducive to more advanced, active learning; both students and lab instructors reported this lab time to be more productive and less frustrating. In this section, we discuss quantitative results from data comparing paired to solo students.

4.1 Success Rate/Retention

First, we examined the percentage of students who succeeded in the class by completing the course with a grade of C or better. Historically, beginning Computer Science classes have poor success rates. Despite the good intentions and diligent work of computer science educators, students find introductory computer science courses very daunting—so daunting that typically one-quarter of the students drop out of the classes and many others perform poorly (by receiving a grade of D or F).

Using the above criteria, we combined results for the Fall 2001 and Spring 2002 semesters as shown in Table 1. Our results indicate that pairing helped the non-CS majors but did not cause any significant improvement among the CS majors. A Chi-Square test was run on the success rates and it showed the solo and paired sections to be statistically independent ($\chi^2(1)=0.0043$, $p < 0.98$). These results are consistent with a similar study at the University of California UC-Santa Cruz that reported 92% of their paired class and 76% of their solo class completed the course [5].

Table 1: Success Rate

Semester	Paired (%)	Solo (%)
Non-CS Majors	**66.4 (N=274)**	**55.9 (N=145)**
CS Majors	**83.0 (N=50)**	**84.0 (N=26)**

(H1) We hypothesized that pair programming would increase the success rate of the students who used the practice (measured by taking students with a grade of C or higher). Our results validated this claim for non-CS majors.

4.2 Performance on Examinations

In the fall semester, students in the paired section performed better on the two-midterm examinations and the final examination, as shown in Table 2. We removed 0 scores from our analysis, making these results based on scores of students who attempted to take the exam.

Table 2: Examination Scores Fall 2001

Exam	Paired Mean	Paired Std Dev	Solo Mean	Solo Std Dev
Midterm 1	**78.7**	11.8	73.4	13.8
Midterm 2	**65.8**	24.2	49.5	27.2
Final	**74.1**	16.5	67.2	18.4

As stated earlier, students chose their class section without knowledge of the experiment or pair programming. We had hoped that their random enrollment in the class would yield equivalent sample groups based on their SAT-Math scores. However, the students in the paired group had a mean SAT-Math score of 662.1 while the solo group had a mean score of 625.4. When using SAT-Math as a covariate, an ANCOVA test does not show any significant difference between sections with regards to any of the exams. Based on these results, we cannot conclude that pair programming in the laboratory helped students perform better on exams. Correspondingly, in the Spring semester we obtained exam results that did not yield any statistically significant improvement in test results by pair programmers. Educators can be concerned that pairs will learn less because they had the ability to lean on their partner. We have certainly not found this to be the case.

(H2) We hypothesized that Students' participation in pair-programming in CS1 will lead to better performance measured by higher scores on the examinations. Our results have not validated this claim to a statistically significant level.

4.3 Performance on Programming Projects

In the fall semester, students in the paired section performed better on the first two of three programming projects, as shown in Table 3.

Table 3: Programming Projects-Fall 2001

Exam	Paired Mean	Paired Std Dev	Solo Mean	Solo Std Dev
Project 1	**94.6**	5.3	78.2	26.5
Project 2	**86.3**	19.7	68.7	33.7
Project 3	73.7	27.1	74.4	29.0

To validate the statistical significance of these results, we ran an ANCOVA test on the data (again examining possible correlation between project scores and the student's SAT-Math scores). The ANCOVA demonstrated a statistically significant improvement in performance of the pairs on Project 1 ($F(1,94)=8.12$, $p<0.0054$) and Project 2 ($F(1,78)=4.52$, $p<0.0367$). However, this analysis did not demonstrate improved performance on Project 3. Perhaps, this is because by Project 3 the lower performing students had dropped in the solo section but were still working in the paired section. In the Spring 2002 semester, we saw no statistically significant difference in project scores by either group, though the paired students often performed marginally better.

(*H3) We hypothesized that students who pair programmed would have higher project scores compared with the solo programmers. From our results, paired and solo programmers have comparable scores in the projects, though in some cases paired programmers have marginally higher scores than the solo students.*

4.4 Results Commentary

We wish to discuss two factors that may influence these results on both the examinations and the projects. First, the implementation of pairing in the lab portion of the course may have enough of a positive influence to keep students from dropping out of the course, or it could have boosted their grades enough to allow them to pass the course. As a result, the poorer performing students may have negatively influenced the calculation results of the paired section. These poorer performing students dropped the class or did not take exams in the solo section, removing themselves from the calculation pool. Researchers at UC-Santa Cruz have also made this same speculation, [5] because their paired section also did not achieve statistically significant higher test scores than the unpaired section. Additionally, only approximately 40% of the exam content required program code to be written in the answers. The rest of the exams were short answer and multiple choices. Quite feasibly, pair programming might not help improve students' answers to short-answer and multiple-choice questions.

4.5 Attitude

Students in paired labs will have a positive attitude toward working in collaborative software development environments. A survey was conducted among the students who worked in pairs throughout the spring semester. Eighty percent of the students in the paired section indicated that they were neutral (19.8%) or positive (59.9%) about pairing in the future.

(H5) We hypothesized that students in paired labs will have a positive attitude towards working in collaborative software development environments. Our survey results supported these claims.

5. Challenges

As with all learning methodologies there were certain challenges we encountered during this experiment over the fall and spring semesters.

- In a small percentage of cases, the random pairing led to incompatible partners, which led to conflicts during working. We hope to address this in our future work by matching people according to personality profile and/or skill type.
- The LIs have to monitor that one partner does not dominate the pair or that one partner is burdened with the entire workload. Student peer evaluations often to not reflect such difficulties. However, to certain degree, students to not want to "turn in" their partner. As a result, the LIs must also be observant of the chemistry and working of the pair in the closed labs

6. Conclusions and Future Work

Our study provides strong results of the following findings:

- Pair programming helps in the retention of more students in the introductory computer science stream.
- Students in paired labs have a more positive attitude toward working in collaborative environments; this should ultimately help the student in his/her professional life.
- Pair programming in an academic environment reduces the burden on the LI because the pairs helped each other, enabling the LI to perform more efficiently.
- From the results we have obtained regarding the tests and the projects, we can conclude significantly that pair programming among students is in no way a deterrent to student performance.

We plan to continue the experiment in the 2002-3 academic year with some modifications. Personality profiles like the Myer-Briggs personality tests will be used to determine a student's personality. We will experiment with successful matching patterns. This will help to provide us with more insight as to how personality profile matters in pair programming. We will also gather results for minority and female students to obtain meaningful results for these important groups.

7. Acknowledgements

The National Science Foundation Grant DUE CCLI 0088178 provided funding for the research in this pair programming experiment.

References

[1] Bevan, J., Werner, L., and McDowell, C., "Guidelines for the User of Pair Programming in a Freshman Programming Class," presented at Conference on Software Engineering Education and Training, Kentucky, 2002.

[2] Cockburn, A. and Williams, L., "The Costs and Benefits of Pair Programming," in *Extreme Programming Examined*, G. Succi and M. Marchesi, Eds. Boston, MA: Addison Wesley, 2001, pp. 223-248.

[3] DeMarco, T. and Lister, T., *Peopleware*. New York: Dorset House Publishers, 1977.

[4] Ferzli, M., Wiebe, E., and Williams, L., "Paired Programming Project: Focus Groups with Teaching Assistants and Students," North Carolina State University, Raleigh, NC CSC TR-2002-16, 2002.

[5] McDowell, C., Werner, L., Bullock, H., and Fernald, J., "The Effect of Pair Programming on Performance in an Introductory Programming Course," presented at ACM Special Interest Group of Computer Science Educators, Kentucky, 2002.

[6] Slavin, R., *Using Student Team Learning*. Boston: The Center for Social Organization of Schools, The Johns Hopkins University, 1980.

[7] Slavin, R., *Cooperative Learning: Theory, Research and Practice*. New Jersey: Prentice Hall, 1990.

[8] Williams, L., Kessler, R., Cunningham, W., and Jeffries, R., "Strengthening the Case for Pair-Programming," in *IEEE Software*, vol. 17, 2000, pp. 19-25.

[9] Williams, L., Wiebe, E., Yang, K., Ferzli, M., and Miller, C., "In Support of Pair Programming in the Introductory Computer Science Course," *Computer Science Education*, vol. September, 2002.

[10] Williams, L., Yang, K., Wiebe, E., Ferzli, M., and Miller, C., "Pair Programming in an Introductory Computer Science Course: Initial Results and Recommendations," presented at OOPSLA Educator's Symposium, Seattle, WA, 2002.

[11] Williams, L. A., "The Collaborative Software Process PhD Dissertation," in *Department of Computer Science*. Salt Lake City, UT: University of Utah, 2000.

A Multi-institutional Study of Peer Instruction in Introductory Computing

Leo Porter[1], Dennis Bouvier[2], Quintin Cutts[3], Scott Grissom[4], Cynthia Lee[5], Robert McCartney[6], Daniel Zingaro[7], and Beth Simon[1]

[1]University of California, San Diego
[2]Southern Illinois University Edwardsville
[3]University of Glasgow
[4]Grand Valley State University
[5]Stanford University
[6]University of Connecticut
[7]University of Toronto Mississauga

ABSTRACT

Peer Instruction (PI) is a student-centric pedagogy in which students move from the role of passive listeners to active participants in the classroom. Over the past five years, there have been a number of research articles regarding the value of PI in computer science. The present work adds to this body of knowledge by examining outcomes from seven introductory programming instructors: three novices to PI and four with a range of PI experience. Through common measurements of student perceptions, we provide evidence that introductory computing instructors can successfully implement PI in their classrooms. We find encouraging minimum (74%) and average (92%) levels of success as measured through student valuation of PI for their learning. This work also documents and hypothesizes reasons for comparatively poor survey results in one course, highlighting the importance of the choice of grading policy (participation vs. correctness) for new PI adopters.

1. INTRODUCTION

Peer Instruction (PI) has gained considerable traction among computer science educators and there have been a number of studies demonstrating its efficacy in a variety of dimensions. Students value PI [5, 8, 10] and learn more in PI classes compared to traditional lecture classes [11, 13]. PI is also associated with low failure rates [6] and increased retention of majors [9].

The vast majority of PI studies in CS take the form of evaluating a single instructor [5, 8, 10] or implementation at a single institution [5, 6, 9]. As such, one concern is that the reported PI results overrepresent those occasions where PI has proven successful. Given the current trend of increased PI adoption, it is important to establish the kinds of outcomes that can be expected across larger datasets and institution types. In addition, it is important to begin studying the ways in which new adopters adopt PI. To what extent is PI adopted wholesale? How are the steps of PI altered to suit the instructor or student demands?

This paper reports on a set of PI adoptions with a broad range of class parameters and types of institutions. Only one other paper has offered such a multi-institution view, but it examined only small classes at private liberal-arts colleges [8]. In the present study, seven instructors and their students, from multiple institutions of different types, were surveyed. The instructors range from new adopters of PI to experienced PI users. The level of success was not uniform across the instructors.

Our key findings are:

- Consistent with previous studies, a supermajority of students in all studied classes liked and would recommend PI.
- Successful PI implementation requires that the instructor's motivations for using PI are clear to students.
- The grading policy attached to in-class PI question responses appears to have an effect on student engagement and satisfaction.

2. RELATED WORK

PI is characterized by asking challenging, in-class conceptual questions of students. For each question, students individually respond, discuss the question in small groups, and respond again based on their new understanding [1]. These questions should target common misconceptions and/or core course concepts. To be most effective, PI requires other supporting course changes. For example, many instructors require additional preparation from students before class in order to make best use of limited class time. This preparation can consist of pre-lecture reading and associated quizzes [1, 14] or clicker quizzes at the start of class [14].

Interactive classrooms, including PI classrooms, have shown significant increases in student learning in physics [4]. In

SIGCSE '16, March 02-05, 2016, Memphis, TN, USA

DOI: http://dx.doi.org/10.1145/2839509.2844642

Table 1: Institution, course, and instructor characteristics. For major, (decl.) and (ant.) denote declared and anticipated, respectively. N/A denotes data not available.

Identifier	N-A	N-B	N-C	E-D	E-E	E-F	E-G
Institution	R1	PUI	PUI	R1	R1	R1	PUI
Size	Large	Large	Large	Large	Large	Large	Small
Public/Private	Public	Public	Public	Public	Public	Public	Private
Course	CS1.5	CS1	CS1	CS1	CS0	CS1	CS1
Language	Java	Java	Java	Python	Alice	Matlab	Java
Times Taught this Course	6	10+	10+	10+	2	1	0
Courses Taught using PI	0	0	0	0[a]	10	3	1
Students Enrolled at End of Course	64	30	36	151	87	98	19
Survey participants	62	29	13	65	87	92	15
Percentage of CS majors	34% (decl.) 59% (ant.)	50%	29% (decl.)	70% (ant.)	1%	6.4%	44% (ant.)
Freshman	3%	40%	36%	>95%	34%	34%	37%
Sophomore	40%	27%	42%	N/A	45%	16%	26%
Junior	37%	27%	13%	N/A	14%	33%	22%
Senior	21%	6%	9%	N/A	8%	17%	16%
Percentage of Students who Previously used Clickers	77%	28%	22%	3%	N/A	31%	25%
Avg. # of PI Questions per class	7	7	6	5	5	4	5
Length of class (min)	75	50	75	60	80[b]	50	80
(D)eveloped or (A)dopted Questions	D	D	D	D	D	D	A

[a] Instructor of course E-D had not taught a class using PI, but had been part of PI research and course development. As such, that instructor is considered experienced in PI.

[b] 30 minutes each week was spent on a practice code-writing quiz.

CS, a number of studies have reported on the success of PI, including improved student satisfaction [5, 8, 10], student learning [7, 15], final exam grades [11, 13], failure rates [6], and retention of majors [9].

The present work provides additional evidence that, for both new and seasoned adopters of PI, students widely laud the change to the course structure. As noted in earlier research [10], it is not always the case that PI is adopted following all recommended practices. The results of the present work lead us to examine ways in which PI is adopted, and we find suggestive evidence that student satisfaction can be significantly impacted by grading clicker results on correctness rather than participation.

3. METHOD

Each of the seven instructors in this study reported the teaching of one introductory programming course. Four of the instructors had experience with PI, either by having taught a course in PI or by having participated significantly in the development of a PI course. The other three instructors were new to PI. We label our courses with two letters: N (novice) or E (experienced) to indicate the instructor's PI experience, and a letter A-G to differentiate each course (assigned, within N and E categories, by decreasing years of teaching experience for the instructor). Course and instructor characteristics are provided in Table 1.

A brief description of each course is provided below.

Course N-A: CS 1.5 - Object-Oriented Design and Programming: This course concentrates on the object-oriented paradigm, particularly encapsulation, inheritance, and polymorphism. Programming assignments emphasize graphics and event-driven interaction.

Course N-B: CS1 – Computer Science I: This course is an introduction to programming and computer science whose topics include: simple and structured data types, program control structures, problem analysis, algorithm design, and implementation using a high-level language (Java).

Course N-C CS1– Introduction to Computing I: This Java course covers types/variables, assignment, conditions, loops, classes/objects, files, and arrays.

Course E-D: CS1: This course is an introduction to procedural programming in Python for CS majors covering basic types, expressions, state, control structures, function definition and use, and lists.

Course E-E CS0 – Fluency in Information Technology: This non-majors computing course is required for all psychology majors and as a general education requirement for a subset of university students (those within a specific "college"). The goals of this "general education" course in computing include computational thinking and communicating and collaborating about computational artifacts (in this case Alice programs and Excel sheets).

Course E-F: CS1 – Introduction to Programming in Matlab: This course is an introduction to MATLAB programming for the Cognitive Science department and uses the Media Computation approach [3]. Students study foundational programming constructs such as data manipulation, conditional statements, for-loops, while-loops, and various types of vector and matrix indexing.

Course E-G: CS1 – Computer Science I: This course is a required course for CS and Mathematics majors taught using Media Computation [3]. Concepts include variables, objects, methods, loops, conditionals, and class design.

Table 2: Student feedback on the value of PI. Percentages reflect student agreement. Agreement values under 80% are highlighted.

Question/Identifier	N-A	N-B	N-C	E-D	E-E	E-F	E-G
Thinking about clicker questions on my own, before discussing with people around me, helped me learn the course material.	68%	86%	100%	94%	95%	95%	100%
Most of the time my group actually discusses the clicker question.	90%	93%	100%	88%	98%	97%	100%
The immediate feedback from clickers helped me focus on weaknesses in my understanding of the course material.	74%	96%	100%	95%	99%	91%	100%
Knowing the right answer is the only important part of the clicker question.*	37%	14%	15%	12%	23%	17%	20%
Generally, by the time we finished with a question and discussion, I felt pretty clear about it.	69%	90%	100%	94%	97%	84%	93%
Clickers helped me pay attention in this course compared to traditional lectures.	58%	93%	100%	95%	90%	90%	100%
Clickers with discussion is valuable for my learning.	74%	93%	100%	100%	94%	91%	93%
I recommend that other instructors use this approach (reading quizzes, clickers, in-class discussion) in their courses.	71%	90%	100%	98%	93%	87%	100%

One instructor, experienced in teaching PI, is known by all instructors and actively assisted the novice instructors in weekly half-hour Skype meetings during their first PI term. The instructors surveyed their students using a common instrument, which enabled comparison of responses across courses.

4. RESULTS

In student self-report surveys, we asked for views on the value of the PI approach in supporting various aspects of the learning experience and views on the instructor's implementation of PI in the classroom.

4.1 Student Perception of the Value of PI

Students reported on their perception of the value of PI (see Table 2). For all but one question (denoted with a *), higher percentages are better. Responses below 80% positive are highlighted. We note two trends in Table 2.

The first trend is that students overwhelmingly value PI. They report that they pay better attention in class, believe it helps them identify weaknesses earlier, and believe the process helps them learn. As a result, the vast majority of students (91% per class on average) recommend that more instructors use PI in their courses. In two classes, 100% of students would recommend that other instructors use PI. These results demonstrate that across a wide range of institutions and instructors, students both value PI and desire that PI be used by more instructors.

The second trend is that the students in Course N-A perceive PI considerably differently than students in other courses. Compared to students in other courses, the students in Course N-A recommend PI less often, felt discussion was less valuable, and generally reported less value from the PI process. Perhaps most striking is the large percentage (37%) of students who believe the value of a clicker question is only in having the correct answer. We will revisit this anomalous result in the Discussion.

4.2 Student Perception of PI Implementation

Table 3 indicates student satisfaction with how PI was implemented regarding difficulty and timing. To express dissatisfaction, students could respond either "too long"/"too difficult" or "too short"/"too easy."

The majority of classes saw high degrees of satisfaction with the PI implementation. Both Courses N-A and E-F stand out as having lower levels of satisfaction with the implementation, but recall that only Course N-A experienced the overall lower value of PI.

Question Difficulty. In Course N-A, some students (23%) felt that questions were too difficult. Only two other courses had more than 4% of students who reported that questions were too difficult: Course E-F with 14% and Course E-G with 13%.

Question Time Allowed. In general, if students were unsatisfied with the time allowed for the initial vote, then they felt that they had too little time (notably in Courses N-A, N-C, E-D, E-E, and E-G) rather than too much time. Course N-A was again an outlier with 21% of students responding that they had too little time relative to 7% who felt they had too much time. Although instructors set the questions (and drive the pace), it is critical for students to be given time to think through the questions on their own. Question design (e.g. word choice, clarity, answer options) can seriously impact student time needed to read the question.

Discussion Time Allowed. An interesting trend appears regarding time allowed for peer discussion. Of those students who were not satisfied with the time allowed for peer discussion in the courses of all four experienced instructors, more students felt that too much time was allowed. A possible reason for this is that experienced PI instructors are more comfortable spending time on peer discussion, and may have personal evidence suggesting the value of providing students with more time to talk among themselves. For novice instructors, time circulating in the classroom or silently standing up-front can be initially unnerving, and at the very least is a change in their teaching style. However, results in Course N-A more closely match those of courses taught by experienced instructors in that unsatisfied students felt that too much time was allowed. As can be observed from Figure 1, this may be related to the comparatively fewer students in Course N-A who report always discussing with their peers: if some students are not discussing, then they are waiting for class to move forward.

Table 3: Student feedback on the implementation of PI. Percentages reflect students responding "OK" or "About right". Values under 80% are highlighted.

Question/Identifier	N-A	N-B	N-C	E-D	E-E	E-F	E-G
From the point of helping me learn, the content of clicker questions was: (too hard, okay, too easy)	76%	83%	100%	88%	94%	78%	80%
In general, the instructor gave us enough time to read and understand the questions before the first vote: (too short, about right, too long)	72%	89%	92%	81%	87%	78%	87%
The amount of time generally allowed for peer discussion was: (too short, about right, too long)	89%	87%	92%	79%	86%	77%	73%
In general, the time allowed for class-wide discussion (after the group vote) was: (too short, about right, too long)	70%	86%	100%	64%	81%	63%	93%

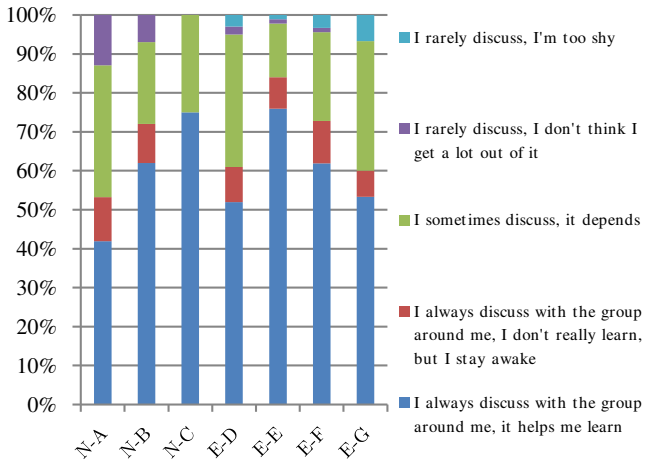


Figure 1: Student responses to the question: Which of the following describes your discussion practices this term?

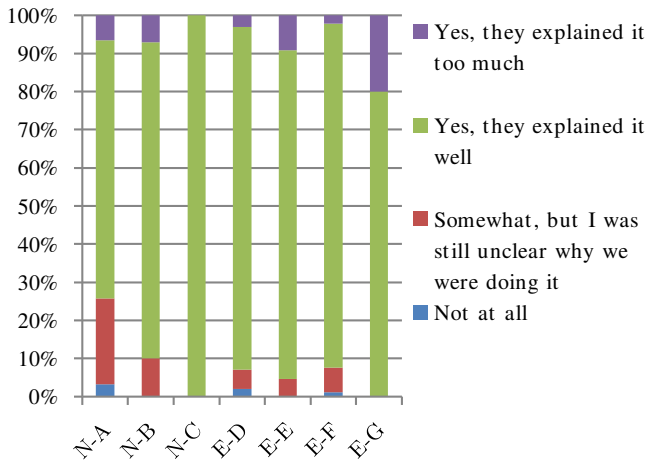


Figure 2: Student responses to the prompt: The professor explained the value of using clickers in this class.

Class-wide Discussion Time Allowed. Regarding class-wide discussions, unsatisfied students felt that they had too much time in Courses N-A, E-D, E-E, and E-F. Course E-F had the most students reporting that too much time was spent on class-wide discussion (26% too much compared with 11% too little). This instructor had little experience with the programming language used in the course; it is possible that this caused a mismatch between where students struggled and where the instructor anticipated struggles.

Student Behavior during Discussion. Figure 1 provides the breakdown of student responses regarding their discussion habits. For all but one course, the majority of students reported valuing the discussion with their group as it helped them learn. For all courses, only a small minority of students either did not discuss because they did not value discussion or because they were too shy.

Explanation of the Purpose of PI. Figure 2 provides the breakdown of student responses regarding the explanation from the instructor on why clickers were being used. For all but Instructor N-A, more than 90% of students thought that the instructor explained the use of clickers well or did so too much. Instructor N-A had a considerably larger percentage of students reporting that they were unclear why they were using clickers. The other anomalous course result was Course E-G, where 20% of students felt that the instructor explained the use of clickers too much.

5. DISCUSSION

5.1 Value to Students and Instructors

Our results suggest that PI can be successfully adopted at a variety of institutions in introductory computing courses. At least 71% of students (avg. 91%) would recommend that other instructors use this approach in their courses.

The seven instructors in this work all reported a dramatic change in their usual classroom experience upon implementing PI. These instructors varied in terms of reasons for adoption, amount and type of adoption support, and teaching experience. Instructors had varying comfort levels with the courses and varying support for developing and implementing clicker questions. In fact, some had no more to go on than a notion of allowing students to "test" their knowledge. Some read publications on its use, some reviewed clicker slides prepared and used in computing courses, and some had the opportunity to TA for or repeatedly observe an experienced PI instructor. *Independent of these background factors, all instructors report that the next time they teach this class, they will teach it with PI.*

This said, one course (N-A) stands out as yielding poor survey results compared to the other courses. We next reflect on why this may be the case.

5.2 Exploring Course N-A Differences

Differences between N-A and the other courses can be seen fairly uniformly in both the valuation and implementation

surveys. Through reflective discussion and our reading of PI reports in other disciplines, we highlight two structural issues for discussion. First, the instructor required correctness for a portion of the PI grade (each question was worth 2 points: 1 for correctness and 1 for participation). Second, a notable 25% of the class felt that the instructor did not sufficiently explain why clickers were being used. Together, we posit that these two issues contributed to a very different classroom culture regarding PI, as evidenced by the markedly different student survey responses. In sum, students in Course N-A (more than those at other institutions):

1. Felt that clicker questions were too hard
2. Felt that too little time was allotted to read and understand questions
3. Did not always discuss with their peers
4. Did not find value in hearing other students provide explanations in class-wide discussion and,
5. Reported lesser perception of the value of PI for their learning.

5.2.1 Grading on Correctness

We suspect that the change from grading on participation to grading on correctness fundamentally changes the atmosphere of the course. Consider the first four points in the above list. This kind of student affect — feelings that questions are too hard and that discussion is not useful — make sense if students perceive PI questions as "standard quiz" questions rather than peer discussion questions. The fact that Instructor N-A gave points for correctness, even though he/she also gave points for participation, may have factored into students perceiving questions as a test of something they should already know, rather than tools designed to build their understanding.

There are two other concerns related to grading on correctness rather than participation. The first is that students become concerned about arguing their interpretation of the question, and this can disrupt the learning process. A focus on learning is hampered when an incorrect answer in the learning process itself has grade-based implications. The second, related issue, is that questions cannot be too difficult or will be viewed as unfair. This is problematic when the recommended correctness range for the individual vote on PI questions is 35-70% [1] and when there is evidence that students benefit considerably more from difficult questions [15]. Instructors who both grade on correctness and recognize the limits that this poses on difficulty may respond in-kind by offering easier questions. This shying away from difficult questions may explain why many students thought that PI was not beneficial for their learning.

We can see evidence of students experiencing Course N-A differently than the other courses through open-ended student responses. For example, a student in Course N-A reports:

In this lecture I am more focused on trying to guess the answers to the questions than on internalizing and understanding the course content. In standard lectures I am focused on taking clear and thorough notes and absorbing the material.
–Student in Course N-A

A quote from a student in Course E-G may provide more insight into the importance of the grading structure, based on their report of clicker use in a different class:

I have another clicker course, this one is far better. In the other course, every clicker question is graded. It feels too much like the professor just wants to play game show host and puts to [sic] much weight on the correct answers and not the process of getting the answer. This is opposite for this course. I feel like participation should just be graded.
–Student in Course E-G

Given this criticism of grading on correctness, it is essential to examine Course E-E. Like Course N-A, Course E-E had a correctness requirement, but it was differently implemented in a manner that reinforced the role of clicker questions in the learning process rather than the assessment process. In Course E-E, students were graded on participation, but they had to get at least 50% of all questions (including individual and group responses) in each lecture period correct to get those participation points. This approach was devised based on experience in an earlier offering of the course, which required only participation and where students were noted answering randomly and engaging in unrelated activities in the classroom. The instructor believed that a change in policy was needed based on the fact that the course was a required, non-majors course. Students generally did not start the course with a great deal of enthusiasm, nor much understanding of what value the course held for them. In explaining the "half correct" policy, the instructor was able to reiterate that clicker questions are for helping students engage in developing expert analysis and argumentation skills. The policy reinforced Instructor E-E's primary course learning goal: getting students to learn how computing people see problems.

A student from Course E-E discusses this policy, noting his/her need to prepare, but expressing satisfaction with the awarding of discussion points:

This class was very different from my other classes as it truly made me be on top of my game. I did like the grading structure and how participation points were fair.
–Student in Course E-E

And some students did appear to get the message that their engagement in reasoning about the question, not simply getting points, was the goal.

[I]n [this course], discussion and proof of understanding is a vital part of lecture. With clicker questions, as a student, I was able to engage in thoughtful reasoning.
–Student in Course E-E

5.2.2 Explaining Pedagogical Change

It is critically important to explain (repeatedly) to students any deviation from expected classroom norms. Students have both experience with and expectations of college classroom learning. They know what happens in a lecture and have techniques that they expect to use in order to learn and measure their progress toward success. From our collective experience, we can report that students claim that they "have to sit at the front of the class" in order to learn/stay awake, want the instructor to "just explain it," or complain that the lack of lecture "forced me to learn it all myself." PI completely pulls the rug out from under the students by challenging them to re-examine their established, comfortable, and often perceived successful learning habits.

In a popular 2-page "Tips for Successful Clicker Use" summary, Douglas Duncan (Univ. of Colorado, Astronomy) lists as his second tip: "You MUST MUST MUST explain to students why you are using clickers. If you don't, they often

assume your goal is to track them like Big Brother, and force them to come to class. Students highly resent this." (emphasis original)[12]. Perhaps more tellingly, the first item on his list of Practices that Lead to Failure is "1. Fail to explain why you are using clickers." [2]

In Course N-A, 74% of students report that their instructor explained to the class why he/she was using clickers. However, the remaining 26% of students who felt that clickers were not explained adequately (or at all) was the highest percentage among the courses. This emphasizes the importance of explaining the value of PI not just once and in not just one way. Moreover, for Course N-A, it is likely that students were especially sensitive to this issue as most students had previously taken an introductory course where the instructor used clickers to take attendance. The presence of these kinds of non-pedagogical uses of clickers only heightens the need to explain the pedagogical goals of PI.

5.2.3 Other Factors

Note that, in addition to the two structural issues we have highlighted, Course N-A also differs from other courses in other ways. The course is a CS 1.5 course (not a CS1 course), many of the students used clickers in the past, and the proportion of freshmen is lower. There is little precedence for these differences contributing to the outlying survey results for this course, however. The earliest experience reports of PI in CS report on successful adoption in CS1.5 courses [10], and we know of no evidence suggesting that any novelty of clickers wears off after a single course.

6. CONCLUSION

In this multi-institutional study of student satisfaction in Peer Instruction (PI) courses, we find further evidence of PI being highly valued by students. We also find that one course yielded lower student satisfaction than the other courses. We have argued that this lower satisfaction may have stemmed from two adoption decisions: grading on correctness, and not convincingly arguing to students why clickers and PI are being used. We offer two conclusions here. First, new adopters of PI can expect levels of success similar to those reported by others with considerable PI teaching and development experience. Second, it is important to evaluate adoption decisions. A pedagogy so widely-applicable as PI will inevitably engender debate over the particulars of day-to-day implementation and interaction with students. We encourage all PI instructors to reflect on and make explicit the reasons undergirding their PI-based decisions so as to maximize the value of PI for students.

7. ACKNOWLEDGEMENTS

Thank you to the reviewers for their helpful suggestions. This work was supported in part by NSF grant 1140731.

8. REFERENCES

[1] C. H. Crouch and E. Mazur. Peer instruction: Ten years of experience and results. *American Journal of Physics*, 69, 2001.

[2] D. Duncan. *Tips for Successful "Clicker" Use*, 2008. Accessed 8/24/15.

[3] M. Guzdial. A media computation course for non-majors. *ACM SIGCSE Bulletin*, 35(3):104–108, 2003.

[4] R. R. Hake. Interactive-engagement vs. traditional methods: A six-thousand-student survey of mechanics test data for introductory physics courses. *American Journal of Physics*, 66(1), 1998.

[5] C. B. Lee, S. Garcia, and L. Porter. Can peer instruction be effective in upper-division computer science courses? *Transactions on Computing Education*, 13(3):12:1–12:22, Aug. 2013.

[6] L. Porter, C. Bailey Lee, and B. Simon. Halving fail rates using peer instruction: A study of four computer science courses. In *Proceeding of the 44th ACM Technical Symposium on Computer Science Education*, 2013.

[7] L. Porter, C. Bailey-Lee, B. Simon, and D. Zingaro. Peer instruction: Do students really learn from peer discussion in computing? In *7th Annual International Computing Education Research Workshop*, 2011.

[8] L. Porter, S. Garcia, J. Glick, A. Matusiewicz, and C. Taylor. Peer instruction in computer science at small liberal arts colleges. In *Proceedings of the 18th ACM Conference on Innovation and Technology in Computer Science Education*, 2013.

[9] L. Porter and B. Simon. Retaining nearly one-third more majors with a trio of instructional best practices in cs1. In *Proceeding of the 44th ACM Technical Symposium on Computer Science Education*, 2013.

[10] B. Simon, M. Kohanfars, J. Lee, K. Tamayo, and Q. Cutts. Experience report: Peer instruction in introductory computing. In *Proceedings of the 41st SIGCSE technical symposium on computer science education*, 2010.

[11] B. Simon, J. Parris, and J. Spacco. How we teach impacts student learning: Peer instruction vs. lecture in cs0. In *Proceeding of the 44th ACM Technical Symposium on Computer Science Education*, 2013.

[12] C. Wieman. *Clicker Resource Guide*. CWSEI - Carl Wieman Science Education Initiative at the University of British Columbia, 2015. Accessed 8/24/15.

[13] D. Zingaro. Peer instruction contributes to self-efficacy in cs1. In *Proceedings of the 45th ACM technical symposium on Computer Science Education*, pages 373–378, 2014.

[14] D. Zingaro, C. Bailey Lee, and L. Porter. Peer instruction in computing: The role of reading quizzes. In *Proceeding of the 44th ACM Technical Symposium on Computer Science Education*, 2013.

[15] D. Zingaro and L. Porter. Peer instruction in computing: The value of instructor intervention. *Computers & Education*, 71:87–96, 2014.

Constructivism in Computer Science Education*

[*Journal of Computers in Mathematics and Science Teaching*, in press.]

Mordechai Ben-Ari
Department of Science Teaching, Weizmann Institute of Science

Constructivism is a theory of learning which claims that students construct knowledge rather than merely receive and store knowledge transmitted by the teacher. Constructivism has been extremely influential in science and mathematics education, but much less so in computer science education (CSE). This paper surveys constructivism in the context of CSE, and shows how the theory can supply a theoretical basis for debating issues and evaluating proposals. An analysis of constructivism in computer science education leads to two claims: (1) students do not have an effective model of a computer, and (2) computers form an accessible ontological reality. The conclusions from these claims are that: (1) models must be explicitly taught, (2) models must be taught before abstractions, and (3) the seductive reality of the computer must not be allowed to supplant construction of models.

Introduction

The dominant theory of learning today is called *constructivism*. This theory claims that knowledge is actively constructed by the student, not passively absorbed from textbooks and lectures. Since the construction builds recursively on knowledge that the student already has, each student will construct an idiosyncratic version of knowledge. To the extent that such knowledge is not identical with 'standard' scientific knowledge, the student is said to have *misconceptions*. Teaching techniques derived from the theory of constructivism are supposed to be more successful than traditional techniques, because they explicitly address the inevitable process of knowledge construction.

Constructivism has been intensively studied by researchers of science education (Glynn, Yeany, & Britton, 1991) and mathematics education (Davis, Maher, & Noddings, 1990), to the extent that "radical constructivism represents the state of the art in epistemological theories for mathematics and science education" (Ernest, 1995, p. 475). However, there has been much less work on constructivism in computer science education.

This article is logically divided into two parts. The first part—after a motivating example—is a survey of the theory of constructivism and its application in science education. The second part of the paper contains my analysis of the theory in the context of computer science and my attempts to apply the theory to issues that are of current interest in CSE.

The discussion will be concentrated within the framework of novice programmers, but constructivist principles are applicable at all levels of computer science education. Given the rapid rate of change of software

*This article is an extended version of a paper was presented at the *Twenty-Ninth SIGCSE Technical Symposium on Computer Science Education*, Atlanta, GA, 1998.

tools and applications, most software engineers in industry and business are continually engaged in education: not only in formal training sessions, but also—perhaps more importantly—in the development of manuals, interfaces and help files. They will find the theory and its applications to be both thought-provoking and relevant to their day-to-day work.

Computer science education (though not perhaps theoretical computer science) probably has more in common with engineering education than with science education. Readers with a background in engineering are invited to speculate about the applicability of my analyses to their fields.

Previous work

There is a large literature on the psychology of programming (Hoc, Green, Samurçay, & Gilmore, 1990; Soloway & Spohrer, 1989; Mayer, 1988); in particular, researchers interested in teaching programming to children or to non-majors are often cognitive psychologists deeply immersed in Piagetian principles. Occasionally, these researchers explicitly acknowledge their commitment to constructivist principles (diSessa, Abelson, & Ploger, 1991, p. 12).

The literature on constructivism in computer science education is in no way comparable with the vast literature in mathematics and physics education. Even today, a search of 'constructivism' in the ACM Digital Library returns only a handful of papers. While many computer science educators have been influenced by constructivism, only recently has this been *explicitly* discussed in published work (Boyle, 1996; Brandt, 1997; Gray, Boyle, & Smith, 1998; Hadjerrouit, 1998).

Motivation

WYSIWYG (What You See Is What You Get) word processors are considered to be the epitome of user-friendliness, because working with them is supposed to be exactly analogous to writing with pen or pencil on a sheet of paper—a routine familiar to everyone who has graduated from elementary school. But consider the following scenario. You type in the title of your term paper, select the text and request boldface font. Unfortunately, as you begin to type the text of the paper, it is also displayed in boldface font! Your pre-existing knowledge of a WYSIWYG word processor is almost certainly the metaphor of ordinary writing which consists of placing blobs of ink sequentially, but arbitrarily, on a sheet of paper (Figure 1). This metaphor cannot furnish an explanation for the phenomenon you have encountered, so you become frustrated, anxious and lose self-confidence.

(Place Figure 1 here.)

Of course, the explanation is trivial: the word processor is not storing blobs of ink, but symbols including implicit symbols for font changes and for indicating the end of a line (Figure 2). (Here we are arbitrarily using HTML notation: ... to delimit boldface font and
 to indicate a line break.) If your selection of the text fragment to change to boldface included an invisible(!) line break character, text typed before the line break will be mysteriously displayed in boldface.

(Place Figure 2 here.)

The correct explanation of WYSIWYG should now be clear. What you *get* is: (1) a data structure for storing text and formatting specifications, and (2) a set of operations on that data structure. What you *see* is: (1) a rendering of the data structure on the screen, and (2) icons and menus to invoke the operations. To *learn* how to use the word processor, you must: (1) create a mental model of the data structure and the effect of each operation, and (2) attribute to each icon and menu item a meaning as an operation.

Constructivism claims each individual *necessarily* creates cognitive structures (models) when learning to use the word processor. Furthermore, it claims that each individual will perform the construction differently, depending on his or her pre-existing knowledge, learning style and personality traits. Hopefully, the

construction is *viable* and the user can successfully use the word processor. Unfortunately, but perhaps inevitably, many users construct non-viable models.

Teaching *how* to do a task can be successful initially, but eventually this knowledge will not be sufficient. As the example tries to show, a student who only knows the procedure for changing from ordinary to boldface font will be helpless when faced with this novel situation. The problem is caused not by stupidity on the part of the novice, nor by incorrectly following the instructions, but by a misconception that is attributable to the lack of a viable model that can explain the behavior of the word processor. The teacher must guide the student in the construction of a viable model so that new situations can be interpreted in terms of the model and correct responses formulated.

The word-processor example illustrates two aspects of learning that are characteristic of computer science. First, since computer science deals with artifacts—programming languages and software, the creator of the artifact employed a very detailed model and the learner must construct a similar, though not necessarily identical, model. Second, knowledge is not open to social negotiation. Given that the word processor is an extant artifact, you cannot argue that its method of using fonts is incorrect, discriminatory, demeaning, or whatever. You may be able to choose another software package, or to request modifications in an existing one, but meanwhile you must learn the existing reality. These two points will be extensively discussed in the rest of the paper.

Epistemology and Constructivism

Educational paradigms

An educational paradigm is composed of four components (Ernest, 1995):

- An *ontology* which is a theory of existence.
- An *epistemology* which is a theory of knowledge, both of knowledge specific to an individual and of shared human knowledge.
- A *methodology* for acquiring and validating knowledge.
- A *pedagogy* which is a theory of teaching.

(See Scheffler (1965) for an introduction to epistemology in the framework of education. Scheffler gives a slightly different decomposition; in particular, he includes *evaluation*: deciding what knowledge is reliable or important.)

We can use this framework to succinctly describe the classical educational paradigm:

- There *is* an ontological reality. Even though scientists accept the theories of relativity and quantum mechanics, the Newtonian model of absolute space and time is the model we generally use for reality. Furthermore, we function as Platonist mathematicians who hold that mathematics has an existence independent of ourselves in which $2 + 2 = 4$ is absolutely true.
- Epistemology is *foundational*. The truth is out there. We come to believe foundations—necessary truths such as $2 + 2 = 4$ and empirical sensory data—and then use valid forms of logical deduction to expand the extent of true knowledge.
- The mind is a *clean slate* that can be filled with knowledge. Once you know enough facts and rules of inference, you can create new knowledge by logical deduction. Carroll (1990) cites the legend of the Nurnberg Funnel which can be used to 'pour' knowledge directly into the learner's head.

- Listening to lectures and reading books are the primary means of *knowledge transmission*. Repetition (drill and practice) will ensure that the knowledge is retained.

The constructivist paradigm is dramatically different:

- Ontological reality is either rejected or at best considered irrelevant. Since we can never truly 'know' anything, ontology cannot influence our educational paradigm.
- The epistemology of constructivism is *nonfoundationalist* and *fallible*. Absolute truth is unattainable, so there is no foundation of truth on which to build. Even $2 + 2 = 4$ is not a necessary truth (Barnes, Bloor, & Henry, 1996, Chapter 7)! Knowledge is constructed by each individual and thus necessarily fallible.
- Knowledge is acquired *recursively*: sensory data is combined with existing knowledge to create new cognitive structures, which are in turn the basis for further construction. Knowledge is also created cognitively by reflecting on existing knowledge. These concepts come from the seminal work of Jean Piaget on the acquisition of knowledge by children; Piaget's work was instrumental in the development of constructivist theories.
- Passive learning will likely fail, because each student brings a different cognitive framework to the classroom, and each will construct new knowledge in a different manner. Learning must be active: the student must construct knowledge assisted by guidance from the teacher and feedback from other students.

Constructivists believe that effective learning demands not just discovery of facts, but the construction of viable mental models, and that teachers must actively *guide* the student in this effort. The task of the teacher in the constructivist paradigm is significantly more difficult than in the classical one, because guidance must be based on the understanding of each student's currently existing cognitive structures.

Note that constructivism does not reject classical means of instruction such as lecturing and reading books. As Mason notes, tongue-in-cheek: "Many educators espousing constructivism have been known to attend lectures on constructivism, and even to have enjoyed them!" (Mason, 1994, p. 197). The problem is not the lecture itself, but the assumption that 'students know what the lecturer told them'. And Mason continues with the suggestion that:

> ... when preparing a lecture, it is the fact of the imminent audience which enables the lecturer to contact the content in fresh ways, in a state conductive to creativity and connection-finding. (Mason, 1994, p. 198)

The concept that the student is trying to construct a model from what are, after all, only words is an appealing theoretical framework for an educator to use in assessing the success or failure of a lecture or other teaching activity.

Conversely, constructivism in not co-extensive with 'modern' teaching methods such as group projects, discovery learning and active tasks. These methods are favored by constructivists *only if* they are designed to enable the students' to build a viable mental model based on pre-existing knowledge. A hands-on activity is useless if "their hands are on, but their heads are out" (Resnick, 1997, p.28).

Constructivism does have a lot in common with *discovery* or *inquiry learning*, where students are expected to discover knowledge by themselves when placed in the appropriate situation. The benefits of discovery are claimed to be:

> ... (1) the increase in intellectual potency, (2) the shift from extrinsic to intrinsic rewards, (3) the learning of the heuristics of discovering, and (4) the aid to conserving memory. (Bruner, 1962, p. 83)

Note that Bruner (1962, p. 85) seems to agree with the constructivist viewpoint that unfettered discovery is not helpful; he distinguishes between *episodic empiricism*, where the student accumulates unconnected facts, and *cumulative constructionism*, where the discovery is organized.

Constructivists differ among themselves as to the relative importance ascribed to the individual learner and to the group in constructing knowledge; these variants are known as *radical* and *social* constructivism, respectively. A discussion of the variants of constructivism is beyond the scope of this article; see Ernest (1995), Phillips (1995).

Constructivism in science education

Studies have shown that relatively few students reach an acceptable level of achievement in high-school science and mathematics (Duit, 1991). Physics teachers seem to have the worst time, as students retain a naive theory of physics despite intensive instruction in Newtonian mechanics (McCloskey, 1983). For constructivists this is not surprising: everyone who has ever thrown a ball—that is, everyone—*knows* that if you don't keep applying force, an object in motion will eventually come to rest. Apparently, these ideas are so entrenched that mere lectures and even experiments have a difficult time evicting them. At most, a certain facility in manipulating formulas is achieved, but this fails as soon as the student attempts to solve a problem that requires deep understanding.

The discrepancy between performance and understanding has also been noted in mathematics education:

> The pupil's fundamental problems with such ideas as negative or complex numbers tend to be overlooked by the teacher mainly because the latter's own implicit beliefs make him or her oblivious to the possibility of somebody having a different ontological stance. ... Another circumstance that helps in concealing ontological difficulties is the fact that a student may become quite skilful in manipulating concepts even without reifying them. (Sfard, 1994, p. 268)

Physics educators are very receptive to constructivist principles. After all, physicists have undergone two massive restructurings of their world within a short period of history: from Aristotelian physics to Newtonian physics and then to Einsteinian physics. One cannot fault them for their reluctance to believe that $E = mc^2$ is an absolute truth. This openness is demonstrated by their willingness to attribute to the student *alternative frameworks* rather than misconceptions.

In fact, von Glasersfeld, a pioneer of constructivism, would never say that something is wrong, because he does not believe in the possibility of establishing universal truths. Instead, he says that concepts are viable "if they prove adequate in the contexts in which they were created" (Glasersfeld, 1995, p. 7). This is analogous to the use of the word in biology to denote an organism adapted to its environment. The box metaphor for variables, and the communications model of reference parameters (discussed below) are simply non-viable, because they cause the student to fail on programming tasks.

According to constructivism, a teacher cannot ignore the student's existing knowledge; instead, he or she must question the student in order to understand exactly what theory the student is currently using, and only then attempt to guide the student to the 'correct' theory. It is perhaps axiomatic for a constructivist that students have consistent theories—they just happen to be at variance with the (currently accepted) scientific theory.

In most fields of science education including computer science, there is a large body of research that catalogs *misconceptions*. A constructivist would view a misconception not as a mistake, but as a logical construction

based on a consistent, though non-standard theory, held by the student. Even Matthews—who is critical of constructivism—is careful to point out that:

> It is with respect to [contemporary physics] that [students] have misconceptions, it is not with respect to the behavior of the natural world. (Matthews, 1994, p. 133)

Merely listing misconceptions is fruitless; a misconception must be accompanied by a description of the underlying model that caused it, and by a suggestion how to base the construction of a viable model on the existing one. Smith III, diSessa, and Roschelle (1993) go so far as to claim that misconceptions form the prior knowledge that is essential to the construction of new knowledge!

It is important not to confuse the use of computers in science education with the study of computer science. Computers are often seen as a tool to increase the constructive content of science education. For example, Hatfield (1991) considers programming, or more generally algorithmics, as constructive. However, his paper is essentially concerned with the contribution of algorithmics to mathematical education, rather than to the constructivist aspects of computer science and programming. Similarly:

> The role of the computer activities is ... to provide an *experiential basis* for all other learning modes. ... the main point is spending the time and effort on the problem, not solving it. (Leron & Dubinsky, 1995, pp. 231, 236)

In CSE, the computer is not just providing an experiential basis, nor is it creating a microworld (Harel & Papert, 1991) in order to facilitate construction of knowledge in another domain. Instead, the students are learning about computing itself—systems, algorithms, languages—and lessons from the use of computers in other fields must be applied carefully.

Criticism of constructivism

Before continuing, we must stress that that there is strong opposition to constructivism. See the articles by Matthews, Nola, Phillips and Ogborn in the Special Issue on Philosophy and Constructivism in Science Education (January 1997) of the journal *Science & Education*. The articles are also available in Matthews (1998).

One critic writes vehemently:

> If radical constructivism is post-epistemological then it is also pre-Copernican and adopts views of science similar to those of the Inquisition that interviewed Galileo. (Nola, 1997, p. 209)

The criticism is not so much of the constructivist theory of learning, but rather of extreme conclusions drawn from constructivist epistemology:

> The one-step argument from the psychological premise (1) "the mind is active in knowledge acquisition," to the epistemological conclusion (2) "we cannot know reality," is endemic in constructivist writing. (Matthews, 1994, p. 151)

Carried to the extreme, radical constructivism leads to *solipsism*, the philosophical claim that the world is one's own mental creation. In turn, this can lead to a rejection of ethics: if the world is my own creation, why should I care what happens to others? Boyle (1996, Section 6.4) takes radical constructivists to task for putting too much emphasis on an individual's cognition at the expense of the biological (Piaget) and social (Vygotsky) foundations upon which cognition must be based.

Carried to the extreme, social constructivism leads to a view of science as a merely political enterprise developed by entrenched elitist groups whose sole purpose is to ensure their own survival. From the fallibility

of scientific knowledge, one slips into relativism of truth, and from the sociology of scientific practice, into demands for empowerment detached from any attempt at objective evaluation of scientific knowledge. The extreme position is stated in the Edinburgh 'strong programme' on the sociology of knowledge (Bloor, 1991; Barnes et al., 1996); for criticism of this position see the articles in Matthews (1998).

The essential question is whether being a constructivist requires an epistemological committment to empiricism and idealism (or social idealism), as opposed to rationalism and realism that seem to come more naturally to scientists. This delicate question can perhaps be avoided by taking the position of 'pedagogical constructivists':

> ... who concentrate solely on pedagogy, and improved classroom practices, ... For [whom], the details of epistemological psychology are unimportant, and not worth disputing about. (Matthews, 1997, p. 8)

Empirical Results in CSE

There is no question that many students find the study of computer science extremely difficult, especially at elementary levels. Before proceeding with a theoretical analysis, it is worthwhile to survey some results that demonstrate the depth of the problem:

- Sleeman, Putnam, Baxter, and Kuspa (1989), Samurçay (1989) and Paz (1996) found that the concept of *variable* is extremely difficult for students. For example, students believe that a variable could simultaneously contain two values, and that after executing `A := B`, the variable `B` no longer contains a value. The students have constructed a *consistent model* using the analogy of a box; the model just happens to be non-viable for successful programming.

- Haberman and Ben-David Kolikant (unpublished research) administered a test designed to check the basic concepts of assignment, read and write statements in Pascal. Given the statements:

  ```
  read(A, B);
  read(B);
  write(A, B, B);
  ```

  many students are not at all sure what happens when you read twice to the same variable or write twice from the variable. They find it difficult to construct a model that identifies 'who' is doing the reading and the writing. Similarly, Samurçay (1989) claims that students' models of `read(A)` may not include the assignment to the variable `A`.

- Madison (1995) used extensive interviews to elicit the internal model of parameters (especially reference parameters) held by students in an introductory course. The students were taught a communications model for parameters, rather than a model of the implementation (copy and reference). The interviews demonstrated that students had constructed consistent, but non-viable, models of the implementation of parameters.

- Similarly, Fleury (1991) discovered 'student-constructed rules' for Pascal parameters that were occasionally successful, but non-viable in the general case.

- Deep misconceptions are not limited to elementary programming. Holland, Griffiths, and Woodman (1997) show the extent of the misconceptions held by students studying object-oriented programming. They found inappropriate conflation of the concept of an object with other concepts like variable, class and textual representation.

- The difficulties that students have in elementary computer science studies are often attributed to the need to spend too much time on the syntax of low-level procedural languages like Pascal and C. But similar phenomena are encountered even when teaching Prolog, a language whose syntax is about as simple as can be imagined. Taylor (1990) studied novice Prolog programmers and found that students constructed models that were not viable:

  > Prolog's behavioral component is complex, and because its syntax is noncommittal, learners are tempted to hallucinate onto it whatever they think appropriate, rather than referring to an interpretation based upon underlying domain knowledge. (Taylor, 1990, p. 308)

- Algorithm and software visualization is an extremely active field of CSE research. Yet Mulholland (1997) found that software visualization in itself does not necessarily help the student unless the visualization is based on a careful analysis of the pedagogic task.

Constructivism in the Context of CSE

To what extent is constructivism applicable to CSE? According to constructivism, students construct knowledge by combining the experiential world with existing cognitive structures. I claim that the application of constructivism to CSE must take into account two characteristics that do not appear in natural sciences:

- A (beginning) computer science student has no *effective model* of a computer.
- The computer forms an *accessible ontological reality*.

By *effective* model, I mean a cognitive structure that the student can use to make viable constructions of knowledge based upon sensory experiences such as reading, listening to lectures and working with a computer. By *accessible ontological reality*, I mean that a 'correct' answer is easily accessible, and moreover, successful performance requires that a normative model of this reality must be constructed. The rest of this section expands on these claims.

The important word is effective. The naive theory of physics held by students is clearly effective, as anyone who has seen professional ball players can testify. They have intuitive models that enable them to implicitly calculate the forces required to achieve superb accuracy when throwing or kicking a ball. Note that diSessa (1988) does not believe that students' intuitive concepts form a well-developed theory. Rather, he claims that they have a large number of fragments called *p-prims*, short for *phenomenological primitives*. This does not materially change the argument, as it is doubtful that intuitive knowledge about computers reaches even the level of diSessa's p-prims.

The empirical results cited earlier (especially the work by Taylor (1990)) show just as clearly that intuitive models of computers are doomed to be non-viable. At most, the model is limited to the grossly anthropomorphic giant brain, hardly a useful metaphor when studying computer science. Pea (1986) gives the name 'superbug' to the idea that a 'hidden mind' within the programming language has intelligence.

At the novice level, the claim is supported by many studies:

> Even if no effort is made to present a view of what is going on 'inside' the learners will form their own. (du Boulay, 1989, p. 285)
>
> ... [we] attribute students' fragile knowledge of programming in considerable part to a lack of a mental model of the computer.... (Perkins, Schwartz, & Simmons, 1988, p. 162)
>
> ... even after a full semester of Pascal, students' knowledge of the conceptual machine underlying Pascal can be very fuzzy. (Sleeman, Putnam, Baxter, & Kuspa, 1988, p. 251)

The lack of an effective, even if flawed, model of a computer can be a serious obstacle to teaching computer science if we accept the claim by Smith III et al. (1993) that prior knowledge, even in the form of misconceptions, is essential to the construction of new knowledge.

Turning now to the question of ontological reality, the computer science student is faced with immediate and brutal feedback on conclusions drawn from his or her mental model. More graphically, alternative frameworks cause bugs. Computer science is unlike school physics: the consequences of misconceptions are exposed immediately, not when you get your homework back a week later. Similarly, from the social viewpoint, there is not much point negotiating models of the syntax or semantics of a programming language.

This claim is based on the fact that almost all introductory computer science instruction includes programming. If, as Dijkstra (1989) suggested, we taught programs as mathematical objects that need not be executed on a computer, the normal constructivist principles would apply. We could talk about the viability of denotational semantics, or the social processes responsible for the belief in the Church-Turing Thesis. If the latter were ever superseded, we would experience a shock no less intense than that experienced by physicists in the early twentieth-century. Clearly, since computer science is unlikely to become a subject that is primarily theoretical, we must generate the motivation to examine our teaching practices without the benefit of an epistemological shock.

The claim cuts at the heart of constructivist *epistemology*, which is nonfoundationalist and fallible. But the pedagogy of constructivism is relatively independent of its epistemology. A physicist has no way of determining if $E = mc^2$ is true, but few of us can resist the temptation to use a computer if it helps us construct knowledge about a language or system. In fact, one of the ultimate tests of your prowess as a computer programmer or software engineer comes when you have to deal with a bug in the underlying hardware, operating system or language compiler. Since you have come to look upon them as ontological reality—as arbiters of truth so to speak—it is extremely difficult to diagnose a problem in the implementation of your mental model, as opposed to a problem in your personal task such as writing a program.

Application of Constructivism in CSE

Many phenomena of CSE can be explained by constructivism:

- The construction of even elementary computer science concepts is haphazard, leading to frustration and to the perception that computer science is hard. This is due to the fact that—in the absence of a viable pre-existing model—models must be self-constructed from the ground up.
- Autodidactic programming experience is not necessarily correlated with success in academic computer science studies. These students, like most physics students, come with firmly held mental models that are not viable for academic studies.
- Graphical user interfaces (GUI) are often touted as ‘intuitive’ and ‘user-friendly’, yet many people earn a comfortable living giving courses to anxiety-ridden users. Icons, scroll bars and menus are merely representations, and seeing a representation alone contributes very little to the construction of a model.
- The reality feedback obtained by working on a computer can be discouraging to students who prefer a more reflective or social style of learning.

In the rest of the paper, I will apply constructivist principles to specific issues in CSE. To avoid misunderstanding, it is important to clarify what is being claimed here. I am not (necessarily) saying that one

approach is superior to another; rather, I am saying that certain conclusions seem to follow directly from constructivist principles, so that if you accept constructivism—which you are not required to do of course—then you must be willing to analyze your teaching methods in light of these conclusions.

GUI and WYSIWYG Angst

Turkle and Papert (1990) wax poetic on the virtues of icons. Yet an icon is just a representation; it is useful only to the extent that the user can *construct* a mental model of object being represented. The icon must undergo *semiosis*: "the process whereby something comes to stand for something else, and thus acquires the status of a sign" (Husén & Postlethwaite, 1994, p. 5411). Today's software packages, both those intended for the general public such as word processors and professional software such as integrated development environments, display dozens of icons. From a semiotic point of view, it may be true that that an icon is better than text, but from a constructivist point of view, what is important is the construction of the model and not the sign that denotes it.

Icons are intuitive to the extent that the analogy between the object shown and the object represented is perfect. But as Glynn (1991) shows, analogies are rarely, if ever, perfect, so one must not lose patience with a novice who has yet to construct a viable model of the underlying machine. For example, consider an icon for the paste operation. The icon is two steps removed from the operation. First, the icon must be deciphered as representing the word paste. (This first step can be skipped if paste is selected from a menu.) Second, the word whose original meaning is 'form a permanent chemical bond between one item and another' must be related to the operation 'insert a copy of the material held in an internal buffer into the current working document at the place pointed to by the cursor'. To understand this operation, you must have a mental model that enables you to understand the four concepts in this sentence. Even if the word 'paste' is avoided, it is hard to see how so many concepts can be contained within an icon.

WYSIWYG (What You See Is What You Get) is another concept that could benefit from constructivist analysis as we showed above. The relevance for CSE is this: courses, help files and tutorials must *explicitly* address the construction of a model, and not limit themselves to behaviorist practices of the form 'to do X, following these steps'. It is a reasonable conjecture that document preparation systems with transparent models like LaTeX and HTML should engender less anxiety among their users than WYSIWYG systems *on complex tasks*. If the underlying model is not accessible, there is a genuine trepidation associated with trying out new or advanced features, for fear that the document will be irrevocably trashed; with a transparent model you can easily insert and then comment-out or remove the explicit commands. Many users of WYSIWYG systems overcome the anxiety and eventually construct viable models, but the anxiety returns as new features are tried or familiar ones used in new contexts. Of course the claims in this paragraph are anecdotal and need empirical verification.

Explicitly Teach the Model

If the student does not bring a preconceived model to class, we must ensure that a viable hierarchy of models is constructed and then refined as learning progresses. This means that the model of a computer—CPU, memory, I/O peripherals—must be *explicitly* taught and discussed, not left to haphazard construction and not glossed over with facile analogies. Furthermore, the choice of language is not arbitrary (as is often claimed) because the "simplicity and visibility of the notional machine can be spoiled by poor language design or implementation" (du Boulay, O'Shea, & Monk, 1989, p. 436).

Teaching the model can be done using diagrams Mayer (1975) or *epistemic games*—formalized procedures for constructing knowledge—such as a model computer (Sherry, 1995) or a notional machine (du Boulay, 1989). Kieras and Bovair (1984) showed that a block diagram of an instrument facilitates the learning of an operational procedure, and Mulholland showed that software visualization (SV) of Prolog programs

is most successful if "there is a clear, simple mapping between the SV and the underlying source code" (Mulholland, 1997). Based on observations of expert programmers and electronics engineers, Petre (1991) believes that declarative reasoning does not really occur; instead, the experts reason operationally in terms of an underlying machine.

An important question is: how detailed should a model be? Does an introductory computer science student have to construct a model in terms of the electronic properties of semiconductors?! The extent and fidelity of the model that must be taught to the students can only be discovered from the experience of teachers of the subject. Sherry's model seems to be too detailed; a better approach is demonstrated by Naps and Stenglein (1996) who created a visualization of a specific concept—parameter passing. Much can be done even with non-computerized epistemic games. For example, take three cheap calculators and attach them to a board (Figure 3), covering up all the non-numeric keys except for `'='`. Each calculator represents one variable and it is possible to practice assignment statements without ever touching a programmable computer.

(Place Figure 3 here.)

Don't Start with Abstractions

My conclusion that a model of the computer be explicitly taught has implications for the teaching of object-oriented programming (OOP) in introductory courses. The abstraction inherent in OOP is essential as a way of forgetting detail, and software development would be impossible without abstraction, but it seems to me that there must be an *object-oriented paradox*: how is it possible to forget detail that you never knew or even imagined? If students find it difficult to construct a viable model of variables and parameters, why should we believe that they can construct a viable model of an object such as a window object? Advocates of an objects-first approach seem to be rejecting Piaget's view that abstraction (or accommodation) *follows* assimilation.

Professional software engineers who use abstractions generally have a fairly good idea of the underlying model. For example, few software engineers have actually written programs for manipulating windows on a screen. But even a general understanding of how images are represented in the computer by bitmaps should be sufficient to enable the engineer to construct a viable model.

I appreciate the attractiveness of an objects-first approach; the gap between the standard libraries (especially the GUI libraries) of a modern programming environment and the model of a computer is so great that motivating beginners has become a serious problem. Furthermore, OOP can be used to teach good software development practice from the beginning because "OOP allows—even encourages—one to address the "big picture" by emphasizing a *strategic* approach to programming" (Decker & Hirshfeld, 1993, p.271).

Turkle and Papert go further and claim that OOP is:

> ... not only more congenial to those who favor concrete approaches, but it also puts an intellectual value on a way of thinking that is resonant with their own. (Turkle & Papert, 1990, p. 155)

This claim is strange, because the point of studying OOP is to learn to *create* abstractions, not just to *use* existing concrete objects. The concreteness of reading and using objects is at most a stepping-stone to modifying, extending and defining them, as advocates of OOP are careful to point out (Decker & Hirshfeld, 1993).

Given these advantages of the objects-first approach, it cannot be dismissed out of hand; on the contrary, the trade-offs probably favor this approach. But if the constructivist viewpoint is valid, teachers of introductory courses that use OOP should be very, very careful not to assume that the students will construct the model that the instructor has, nor even to assume that they will construct a viable model at all.

This viewpoint is supported by the literature on teaching OOP:

- While Adams (1996) opposes deferring the teaching of OOP until late in the curriculum by which time it is difficult to cure students of the low-level paradigms they have developed, neither does he believe that OOP should be taught first when the students are not mature enough to master the *concepts* involved:

  > CS1 novices do not have the cognitive framework to grasp the concepts underlying object-oriented design, because they have no experience dealing with types and functions, much less classes, function members or inheritance. (Adams, 1996, p.79)

  He advocates a middle road where objects are introduced early but only after sufficient procedural programming has been learned to provide an underlying mental model.

- Wolz and Conjura (1994) propose a three-tiered model for teaching introductory computer science which includes mathematical theory (unusual but refreshing!), implementation and mechanical trivia. They report that teaching OOP using C++ in CS2 is successful because students are able to build on previous knowledge learned from CS1: expressing algorithms procedurally in Scheme. On the other hand, they claim that:

  > There is no reason that students in a first course can't learn to use [data types such as queues, stacks, lists, trees and graphs] before learning how they are implemented. (Wolz & Conjura, 1994, p.224)

  From a constructivist point of view, one must evaluate the mental models these students construct; if they are non-viable, they can impede further study.

- Holland et al. (1997) summarize students' misconceptions in an introductory course that uses OOP. Many of these misconceptions are due to conflation of concepts (object/variable, object/class) that can be attributed to the lack of an effective mental model. Based on experience in other disciplines of science education, cataloging and analyzing misconceptions will not be sufficient to improve students' understanding. Instead, research must be done to identify the mental models that cause these specific misconceptions, and guidelines must be developed so that teachers can diagnose and correct the problems.

For an objects-first approach to work, teachers will have to develop ways of explaining the underlying models without destroying the abstractions. My current belief is that introductory CSE should be based on the functional or logic programming paradigm, not only because these languages minimize mechanical trivia, but also (and primarily) because the underlying models can be explained in relatively high-level, hardware-free terms.

Bricolage

Bricolage is a term coined by the anthropologist Claude Lévi-Strauss, who used it in a derogatory sense for the 'science of the concrete' in primitive societies, as opposed to abstract European science. Turkle and Papert (1990) transferred the concept to the context of learning to program, and vehemently defend it as a learning style as valid as the normative 'planning' style that we attempt to teach. This is consistent with a constructivist view of education: different students will approach the construction of knowledge in different ways, and the educational environment must be supportive of these differences.

The manifestation of bricolage in computer science is endless debugging: try it and see what happens. While we all practice a certain amount of bricolage and while concrete thinking can be especially helpful—if not

essential—for students in introductory courses, bricolage is not an effective methodology for professional programming, nor an effective epistemology for dealing with the massive amount of detailed knowledge must be constructed and organized in levels of abstraction (cf. object-oriented programming). The normative planning style that we call software engineering must eventually be learned and practiced.

This belief is likely to be shared by anyone who has studied or worked on non-deterministic systems involving concurrency, real-time or communications, subjects that are simply not amenable to bricolage and can be mastered only through abstract techniques. Students who excel at bricolage often cannot make the transition to master the thought patterns and methods required by these systems. This claim has implications for counselling students. If software development is ultimately about abstraction, a students incapable of or uncomfortable with abstract thought should be discouraged from studying for the profession of software engineer.

Gender

Turkle and Papert (1990) published their article arguing for tolerance of concrete thinking in a journal subtitled *Women in Culture and Society*, and they chose two women to exemplify college students who are concrete thinkers. Since the concrete way of thinking advocated by Turkle and Papert can only go so far in computer science, their coupling of a learning style with a gender stereotype would lead to the unacceptable conclusion that women are not suited for careers as computer scientists.

On the other hand, constructivism—especially social constructivism—has much to say about the task of the teacher and the role of peers in education, and the theory can contribute to the analysis of the well-documented social difficulties faced by women in the computer science classroom and laboratory.

Minimalism

Minimalism (Carroll, 1990, 1998) is an approach to instruction that arose in the design of manuals for software documentation. It is apparently little known outside of this community. (For a good introduction see Van der Meij and Carroll (1998).) The minimalist approach to training and documentation can be summarized as follows:

> ...(1) allowing learners to start immediately on meaningful realistic tasks, (2) reducing the amount of reading and other passive activity in training, and (3) helping to make errors and error recovery less traumatic and more pedagogically productive. (Carroll, 1990, p. 7)

Minimalism has much in common with constructivism as explicitly noted by Van der Meij (1992, p. 7) and Carroll and Van der Meij (1998, p. 84):

- A preference for active learning to enable the student to construct mental models.
- Recognition of the importance of pre-existing knowledge.
- The employment of the inevitable errors and misconceptions as a pedagogical device rather than as a symptom of failure.

Minimalism seems to part company with constructivism in its emphasis—even insistence—on eliminating conceptual material, or at least on deferring it as long as possible:

> It is quite common for training manuals to present a "welcome to the system" preface, a conceptual model of how the system works, ... And none of this, even in the end, does much to facilitate the user's desire to get started on meaningful activity. Rather, it obstructs this goal. (Carroll, 1990, p. 80)

The success of minimalism has been empirically demonstrated in straightforward training tasks like learning to use a word processor. But once the user needs to go beyond elementary tasks, the absence of a viable mental model means that the user's attempts to master advanced material will be frustrating and lead to a reluctance to learn new concepts.

To test this conjecture, I performed an experiment which required the subjects to modify documents in Microsoft Word (Ben-Ari, 1999). The tasks were chosen to be easy if you understand the underlying *concepts*, but quite difficult if you do not. The (sophisticated) subjects almost invariably used bricolage. They restricted themselves to elementary techniques learned in a minimalist setting—behaviorist explanations from colleagues—and made no attempt to investigate the concepts or even to use the Help facility.

Some authors now claim that the dismissal of conceptual material by naive minimalism was mistaken and some way must be found to strike a balance. See the articles by Rosenbaum, Hackos, Redish, Farkas and Draper in the retrospective volume by Carroll (1998). For example:

> ... a manual must: Help users grasp the big picture of the product, that is, help users develop a mental model that helps them predict what to do. (Redish, 1998, p. 240).

Given the empirically proven success of minimalism in the narrow field of technical documentation, it would be interesting to explore a closer integration of minimalist writing techniques with constructivist teaching techniques.

Don't run to the computer

Constructivism suggests that programming exercises should be delayed until class discussion has enabled the construction of a good model of the computer. Too often students become infatuated with the absolute ontology supplied by the computer. Premature attempts to write programs lead to bricolage and delay the development of viable models. While formal methods in computer science education are extremely important, you need not go to the extreme that Dijkstra advocates and entirely give up compilation and execution of programs. There is nothing wrong with experimentation and bricolage-style debugging, as long as it supplements, rather than supplants, planning and formal methods.

Unfortunately, computer science education is heavily weighted on the side of bricolage. A high-school course we are developing comes in for scathing criticism from many students (and some teachers!) because we insist on 'wasting time' on algorithm development and analysis, instead of just getting on with writing and debugging programs.

Laboratory organization

One of the debates in CSE concerns the choice between closed labs—where students work on assignments at an appointed time in a supervised setting, and open labs—where students work on assignments whenever convenient. From a constructivist viewpoint, especially from a social constructivist one, closed labs should be preferable, not only because they soften the brutality of the interaction with the computer, but also because they facilitate the social interaction that is apparently necessary for successful construction. In fact, Thweatt (1994) found empirical evidence for the superiority of closed labs over open labs.

The type of problems assigned is also important; as opposed to minimalism's emphasis on task performance, problems should encourage cognitive operations such as reflection and exploration:

> Another common failing in lab design is to make every task so constrained and explicit that students never need to think about what techniques to use. ... The production of an ill-structured problem is likely to add an element of reality to the lab, and allows the students to have

their own Eureka!s about the underlying nature of the exercise. (Fekete & Greening, 1996, pp. 295, 298)

Assessment

Performance on a test is a poor guide to the students' construction of the rich conceptual models of computer science. A student's failure to construct a viable model is a failure of the educational process, even if the failure is not immediately apparent. Furthermore, in the case of group work, performance-based assessment can mask the misconceptions of individual students (Sleeman et al., 1988). Ideally, constructivist-inspired assessment would be based on an instructor's observation and questioning of students engaged in an unconstrained activity such as a lab project. Unfortunately, this is almost always impractical, and instructors must attempt to designed written questions that elicit information about the student's mental model rather than about the contents of his or her factual memory.

Implications for Research

In their book, Maykut and Morehouse (1994) claim that practitioners of qualitative research must understand its philosophical underpinnings, which are essentially constructivist in nature. The claim can be turned around: a researcher working from a constructivist viewpoint should use qualitative methods.

We are now starting to see more empirical research in CSE done using qualitative methods (Madison, 1995; Mulholland, 1997). These techniques which elicit the internal structures of the student are far more helpful than research that measures performance alone and then draws conclusions on the success of a technique.

As computer literacy becomes common, if not universal, students will begin their academic studies with an effective model of a computer. Research must be done to determine if these models are stepping-stones to the construction of effective models, or obstacles like naive physics.

A Guide for Educators

To summarize the paper, here is a guide for educators on the practical application of constructivism.

- Regardless of your teaching technique (lectures, labs, assignments), you must articulate to yourself the cognitive change that you wish to bring about in the students and structure the activity to achieve this aim. Merely transferring knowledge is not a meaningful aim.
- You must dig underneath your own expert knowledge to expose the prior knowledge needed to construct a viable model of the material that you are teaching. You must ensure that that the students have this prior knowledge.
- In any particular course you will be teaching a specific level of abstraction; you must explicitly present a viable model one level beneath the one you are teaching.
- When a student makes a mistake or otherwise displays a lack of understanding, you must assume that the student has a more-or-less consistent, but non-viable, mental model. Your task as a teacher is to elicit this model and guide the student in its modification.
- You must provide as much opportunity as possible for individual reflection (for example, analysis of errors) and social interaction (for example, group labs).

Clearly, each educator must decide how to apply these aphorisms in a concrete situation.

Conclusion

My analysis of constructivism has led me to conclude that the epistemology of computer science is significantly different than that of, say, physics. Nevertheless, the basic tenet of the theory—that knowledge is constructed by the student—applies to computer science, and its central implication is that models must be explicitly taught.

Given the central place of constructivist learning theory and its influence on pedagogy, computer science educators should study the theory, perform research and analyze their educational proposals in terms of constructivism. Software and language designers should be guided by constructivist principles, though the individuality of the construction by learners implies that no system will ever be universally easy-to-learn, and we educators must learn how to teach these extant artifacts.

Acknowledgements

I would like to thank Abraham Arcavi, Yifat Ben-David Kolikant, Tom Boyle, Bat-Sheva Eylon, Ann Fleury, Sandra Madison and the referees for their critiques of drafts of this article.

References

Adams, J. C. (1996). Object-centered design: A five-phase introduction to object-oriented programming in CS 1–2. *SIGCSE Bulletin*, *28*(1), 78–82.

Barnes, B., Bloor, D., & Henry, J. (1996). *Scientific knowledge: A sociological analysis.* Chicago, IL: University of Chicago Press.

Ben-Ari, M. (1999). Bricolage forever! In *Eleventh workshop of the psychology of programming interest group* (pp. 53–57). Leeds, UK.

Bloor, D. (1991). *Knowledge and social imagery (second edition).* Chicago, IL: University of Chicago Press.

Boyle, T. (1996). *Design for multimedia learning.* Hemel Hempstead: Prentice-Hall.

Brandt, D. S. (1997). Constructivism: Teaching for understanding of the Internet. *Communications of the ACM*, *40*(10), 112–117.

Bruner, J. S. (1962). *On knowing: Essays for the left hand.* Cambridge, MA: Harvard University Press.

Carroll, J. M. (1990). *The Nurnberg Funnel: Designing minimalist instruction for practical computer skill.* Cambridge, MA: MIT Press.

Carroll, J. M. (Ed.). (1998). *Minimalism beyond the Nurnberg Funnel.* Cambridge, MA: MIT Press.

Carroll, J. M., & Van der Meij, H. (1998). Ten misconceptions about minimalism. In J. M. Carroll (Ed.), *Minimalism beyond the Nurnberg Funnel* (pp. 55–90). Cambridge, MA: MIT Press.

Davis, R. B., Maher, C. A., & Noddings, N. (Eds.). (1990). *Constructivist views of the teaching and learning of mathematics.* Reston, VA: National Council for the Teaching of Mathematics.

Decker, R., & Hirshfeld, S. (1993). Top-down teaching: Object-oriented programming in CS 1. *SIGCSE Bulletin*, *25*(1), 270–273.

Dijkstra, E. W. (1989). On the cruelty of really teaching computer science. *Communications of the ACM*, *32*(12), 1398–1404.

diSessa, A. A. (1988). Knowledge in pieces. In G. Forman & P. B. Pufall (Eds.), *Constructivism in the computer age* (pp. 49–70). Hillsdale, NJ: Lawrence Erlbaum Associates.

diSessa, A. A., Abelson, H., & Ploger, D. (1991). An overview of Boxer. *Journal of Mathematical Behavior*, *10*, 3–15.

du Boulay, B. (1989). Some difficulties of learning to program. In E. Soloway & J. C. Spohrer (Eds.), *Studying the novice programmer* (pp. 283–299). Hillsdale, NJ: Lawrence Erlbaum Associates.

du Boulay, B., O'Shea, T., & Monk, J. (1989). The black box inside the glass box: Presenting computing concepts to novices. In E. Soloway & J. C. Spohrer (Eds.), *Studying the novice programmer* (pp. 431–446). Hillsdale, NJ: Lawrence Erlbaum Associates.

Duit, R. (1991). Students' conceptual frameworks: consequences for learning science. In S. M. Glynn, R. H. Yeany, & B. K. Britton (Eds.), *The psychology of learning science* (pp. 65–85). Hillsdale, NJ: Lawrence Erlbaum Associates.

Ernest, P. (Ed.). (1994). *Constructing mathematical knowledge: Epistemology and mathematics education.* London: The Falmer Press.

Ernest, P. (1995). The one and the many. In L. P. Steffe & J. Gale (Eds.), *Constructivism in education* (pp. 459–486). Hillsdale, NJ: Lawrence Erlbaum Associates.

Fekete, A., & Greening, A. (1996). Designing closed laboratories for a computer science course. *SIGCSE Bulletin*, *28*(1), 295–299.

Fleury, A. E. (1991). Parameter passing: The rules the students construct. *SIGCSE Bulletin*, *23*(1), 283–286.

Glasersfeld, E. von. (1995). A constructivist approach to teaching. In L. P. Steffe & J. Gale (Eds.), *Constructivism in education* (pp. 3–15). Hillsdale, NJ: Lawrence Erlbaum Associates.

Glynn, S. M. (1991). Explaining science concepts: a teaching-with-analogies model. In S. M. Glynn, R. H. Yeany, & B. K. Britton (Eds.), *The psychology of learning science* (pp. 219–240). Hillsdale, NJ: Lawrence Erlbaum Associates.

Glynn, S. M., Yeany, R. H., & Britton, B. K. (Eds.). (1991). *The psychology of learning science.* Hillsdale, NJ: Lawrence Erlbaum Associates.

Gray, J., Boyle, T., & Smith, C. (1998). A constructivist learning environment implemented in Java. *SIGCSE Bulletin*, *30*(3), 94–97.

Hadjerrouit, S. (1998). A constructivist framework for integrating the Java paradigm into the undergraduate curriculum. *SIGCSE Bulletin*, *30*(3), 105–107.

Harel, I., & Papert, S. (Eds.). (1991). *Constructionism.* Norwood, NJ: Ablex.

Hatfield, L. L. (1991). Enhancing school mathematical experience through constructive computing activity. In L. P. Steffe (Ed.), *Epistemological foundations of mathematical experience* (pp. 238–259). New York, NY: Springer-Verlag.

Hoc, J., Green, T., Samurçay, R., & Gilmore, D. (1990). *Psychology of programming.* London: Academic Press.

Holland, S., Griffiths, R., & Woodman, M. (1997). Avoiding object misconceptions. *SIGCSE Bulletin*, *29*(1), 131–134.

Husén, T., & Postlethwaite, T. N. (Eds.). (1994). *The international encyclopedia of education.* Oxford: Pergamon.

Kieras, D. E., & Bovair, S. (1984). The role of a mental model in learning to operate a device. *Cognitive Science*, *8*, 255–273.

Leron, U., & Dubinsky, E. (1995). An abstract algebra story. *American Mathematical Monthly*, *102*(3), 227–242.

Madison, S. K. (1995). *A study of college students' construct of parameter passing: Implications for instruction.* Unpublished doctoral dissertation, U. of Wisconsin.

Mason, J. (1994). Enquiry in mathematics and mathematics education. In P. Ernest (Ed.), *Constructing mathematical knowledge: Epistemology and mathematics education* (pp. 190–200). London: The Falmer Press.

Matthews, M. R. (1994). *Science teaching: The role of history and philosophy of science.* New York, NY: Routledge.

Matthews, M. R. (1997). Introductory comments on philosophy and constructivism in science education. *Science & Education*, *6*(1-2), 5–14.

Matthews, M. R. (Ed.). (1998). *Constructivism in science education.* Dordrecht: Kluwer Academic Publishers.

Mayer, R. E. (1975). Different problem-solving competencies established in learning computer programming with and without meaningful models. *Journal of Educational Psychology*, *67*(6), 725–734.

Mayer, R. E. (Ed.). (1988). *Teaching and learning computer programming.* Hillsdale, NJ: Lawrence Erlbaum Associates.

Maykut, P., & Morehouse, R. (1994). *Beginning qualitative research.* London: The Falmer Press.

McCloskey, M. (1983). Naive theories of motion. In D. Gentner & A. L. Stevens (Eds.), *Mental models* (pp. 299–323). Hillsdale, NJ: Lawrence Erlbaum Associates.

Mulholland, P. (1997). *Using a fine-grained comparative evaluation technique to understand and design software visualization tools.* Paper presented at the Empirical Studies of Programmers: Seventh Workshop.

Naps, T. L., & Stenglein, J. (1996). Tools for visual exploration of scope and parameter passing in a programming languages course. *SIGCSE Bulletin*, *28*(1), 295–299.

Nola, R. (1997). Book review of Kenneth Tobin (ed.), The practice of constructivism in science education. *Science & Education*, *6*(1-2), 197–201.

Paz, T. (1996). *Computer science for vocational high-school studuents: Processes of learning and teaching.* Unpublished master's thesis, Technion—Israel Institute of Technology. (in Hebrew)

Pea, R. D. (1986). Language-independent conceptual "bugs" in novice programming. *Journal of Educational Computing Research*, *2*(1), 25–36.

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

Petre, M. (1991). *Shifts in reasoning about software and hardware systems: do operational models underpin declarative ones?* Paper presented at the Psychology of Programming Interest Group Workshop.

Phillips, D. (1995). The good, the bad, and the ugly: The many faces of constructivism. *Educational Researcher*, *24*(7), 5–12.

Redish, J. (1998). Minimalism in technical communication: Some issues to consider. In J. M. Carroll (Ed.), *Minimalism beyond the Nurnberg Funnel* (pp. 219–245). Cambridge, MA: MIT Press.

Resnick, M. (1997). *Turtles, termites, and traffic jams: Explorations in massively parallel microworlds.* Cambridge, MA: MIT Press.

Samurçay, R. (1989). The concept of variable in programming: Its meaning and use in problem-solving by novice programmers. In E. Soloway & J. C. Spohrer (Eds.), *Studying the novice programmer* (pp. 161–178). Hillsdale, NJ: Lawrence Erlbaum Associates.

Scheffler, I. (1965). *Conditions of knowledge: An introduction to epistemology and education.* Chicago, IL: University of Chicago Press.

Sfard, A. (1994). Mathematical practices, anomalies and classroom communications problems. In P. Ernest (Ed.), *Constructing mathematical knowledge: Epistemology and mathematics education* (pp. 248–273). London: The Falmer Press.

Sherry, L. (1995). A model computer simulation as an epistemic game. *SIGCSE Bulletin*, *27*(2), 59–64.

Sleeman, D., Putnam, R. T., Baxter, J. A., & Kuspa, L. (1988). An introductory Pascal class: A case study of student errors. In R. E. Mayer (Ed.), *Teaching and learning computer programming* (pp. 237–257). Hillsdale, NJ: Lawrence Erlbaum Associates.

Sleeman, D., Putnam, R. T., Baxter, J. A., & Kuspa, L. (1989). A summary of misconceptions of high school Basic programmers. In E. Soloway & J. C. Spohrer (Eds.), *Studying the novice programmer* (pp. 301–314). Hillsdale, NJ: Lawrence Erlbaum Associates.

Smith III, J. P., diSessa, A. A., & Roschelle, J. (1993). Misconceptions reconceived: A constructivist analysis of knowledge in transition. *The Journal of The Learning Sciences*, *3*(2), 115–163.

Soloway, E., & Spohrer, J. C. (Eds.). (1989). *Studying the novice programmer.* Hillsdale, NJ: Lawrence Erlbaum Associates.

Steffe, L. P., & Gale, J. (Eds.). (1995). *Constructivism in education.* Hillsdale, NJ: Lawrence Erlbaum Associates.

Taylor, J. (1990). Analysing novices analysing Prolog: What stories do novices tell themselves about Prolog. *Instructional Science*, *19*, 283–309.

Thweatt, M. (1994). CS1 closed lab vs. open lab experiment. *SIGCSE Bulletin*, *26*(1), 80–82.

Turkle, S., & Papert, S. (1990). Epistemological pluralism: Styles and cultures within the computer culture. *Signs: Journal of Women in Culture and Society*, *16*(1), 128–148.

Van der Meij, H. (1992). A critical assessment of the minimalist approach to documentation. In *Tenth annual ACM conference on systems documentation (SIGDOC92)* (pp. 7–17). Ottawa, Canada.

Van der Meij, H., & Carroll, J. M. (1998). Principles and heuristics for designing minimalist instruction. In J. M. Carroll (Ed.), *Minimalism beyond the Nurnberg Funnel* (pp. 19–53). Cambridge, MA: MIT Press.

Wolz, U., & Conjura, E. (1994). Integrating mathematics and programming into a three tiered model for computer science education. *SIGCSE Bulletin*, *26*(1), 223–227.

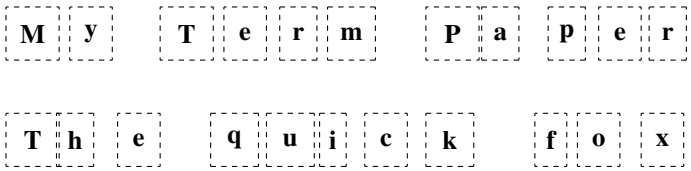


Figure 1: What you (think you) see

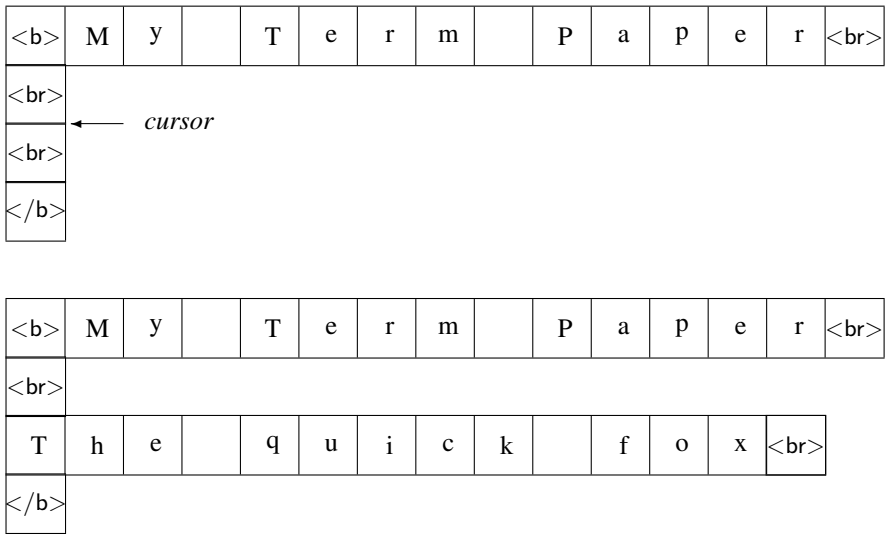


Figure 2: What you (really) get

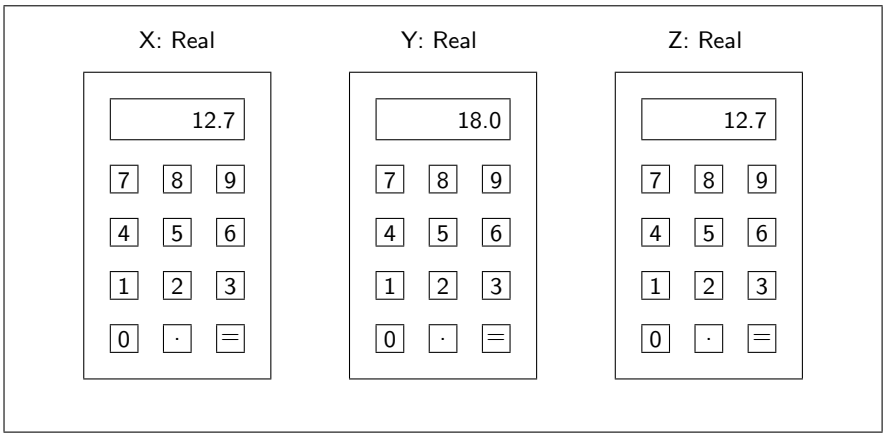


Figure 3: An epistemic game for studying variables

Using Software Testing to Move Students from Trial-and-Error to Reflection-in-Action

Stephen H. Edwards
Virginia Tech, Dept. of Computer Science
660 McBryde Hall, Mail Stop 0106
Blacksburg, VA 24061 USA
+1 540 231 5723
edwards@cs.vt.edu

ABSTRACT

Introductory computer science students rely on a *trial and error* approach to fixing errors and debugging for too long. Moving to a *reflection in action* strategy can help students become more successful. Traditional programming assignments are usually assessed in a way that ignores the skills needed for reflection in action, but software testing promotes the hypothesis-forming and experimental validation that are central to this mode of learning. By changing the way assignments are assessed—where students are responsible for demonstrating correctness through testing, and then assessed on how well they achieve this goal—it is possible to reinforce desired skills. Automated feedback can also play a valuable role in encouraging students while also showing them where they can improve.

Categories and Subject Descriptors

K.3.2 [**Computers and Education**]: Computer and Information Science Education; D.2.5 [**Software Engineering**]: Testing and Debugging—testing tools.

General Terms

Verification

Keywords

Pedagogy, test-driven development, CS1, extreme programming, automated grading.

1. INTRODUCTION

Despite our best efforts as educators, student programmers continue to develop misguided views about their programming activities, particularly during freshman and sophomore courses:

- Once the compiler accepts my code without complaining, I have removed all the errors.
- Once my code produces the output I expect on a test value or two, it will work well all the time.
- My code looks "correct" to me. If it produces the wrong answer, that does not make sense, so there must be something hidden that I do not understand about my code. I will try switching around a few things to see if I can make the problem go away.
- Once my code gives the correct answer for the instructor's sample data, I am finished.

While many computer science students acquire a more balanced view of software development as they learn, other students do not reach such a perspective for many semesters, and some never do so. This situation places both the student and the educator at a significant disadvantage. Anecdotally, many educators report difficulties along these lines [12, 8, 5].

Computer science students will be more successful at learning if they move from this *trial and error* approach to practicing *reflection in action*. "Reflection in action," as originally described by Schön [13], is a characterization of how practitioners complete tasks in the face of uncertainty and novelty. When a technique or part of a solution fails to work, difficulties or confusion cause the practitioner to switch to a reflective mode, examining both the phenomenon at hand and also prior understandings that may have been implicit in his or her behavior. From this reflection, the practitioner then "carries out an experiment which serves to generate both a new understanding of the phenomenon and a change in the situation" [13]. This on-going experimentation is central to finding a viable solution when past experiences do not work in a new context without modification.

Many educators would agree that steering students toward reflection in action is a desirable goal, but typical programming assignments are poor devices for promoting this behavior. Students receive feedback only on the end result they produce and tend to equate a program that "produces the right output" with an "effective solution." The learning process matters little in grade outcomes, and students only receive indirect feedback on what and how they learn via comments on their final solution. Students are often able to succeed at simpler CS1 and CS2 assignments using a trial-and-error approach, which only reinforces a strategy that will handicap their performance in more advanced courses.

This situation can be improved through careful use of *software testing* in programming assignments. From the very first programming activities in CS1, a student should be given the responsibility of demonstrating the correctness of *his or her own* code. Such a student is required to submit test cases for this purpose along with the code. While coding design and style are typically

SIGCSE'04, March 3–7, 2004, Norfolk, Virginia, USA.

assessed using an independent reading of the source code, we must change the way we assess program correctness. Rather than assessing student performance on *whether their programs produce the correct output*, students should be meaningfully assessed on *how well they have demonstrated the correctness of their program* through testing, that is, how correctly and thoroughly their tests conform to the problem.

2. WHY STUDENTS STICK WITH TRIAL AND ERROR

Trial and error is a well-established technique for beginners in any discipline, and it is no surprise that this is where students start out. But why do students persist in this practice long after it becomes a handicap? Buck and Stucki describe one possible reason [4, 5]: most undergraduate curricula focus on developing program *application* and *synthesis* skills (i.e., writing code), primarily acquired through hands-on activities. In addition, students must master basic *comprehension* and *analysis* skills. Without these skills, they are poorly equipped for any strategy beyond trial and error.

Bloom's taxonomy describes six increasing levels of cognitive development that can be used to frame and organize learning objectives, labeled in increasing order of sophistication as: knowledge, comprehension, application, analysis, synthesis, and evaluation. Buck and Stucki provide a concise description of Bloom's taxonomy in a CS education context [4]. Bloom's work suggests that students must master basic comprehension and analysis skills as a prerequisite for effective program writing. Students must develop their abilities in reading and comprehending source code, envisioning how a sequence of statements will behave, and predicting how a change to the code will result in a change in behavior. Yet typical undergraduate curricula focus first and foremost on *writing programs*: application and synthesis skills.

Many educators try to foster comprehension and analysis abilities through code reading assignments or requiring students to manipulate and reason about non-code artifacts [12]. Buck and Stucki propose an "inside/out" pedagogy for introducing CS1 concepts in a manner inspired by Bloom's levels [4, 5]. While this is a powerful approach in organizing assignments, their focus has been on appropriately situating code writing tasks in a context that constrains and directs students as they learn. Others have added small analytical tasks to regular lab assignments [6].

To advance to reflection in action, however, students need more than just an ability to predict how changes in code will result in changes in behavior. In addition, they need continually reinforced practice in hypothesizing about the behavior of their programs and then experimentally verifying (or invalidating) their hypotheses. Further, students need frequent, useful, and immediate feedback about their performance, both in forming hypotheses and in experimentally testing them.

These activities are at the heart of software testing. To write an effective test, students must do more than just come up with a sequence of code actions—they must also hypothesize what resulting behavior they expect. Yet, in most mainstream CS curricula, students get little feedback on their performance in this area. This idea is complementary to Buck and Stucki's focus on the middle levels of Bloom's taxonomy—without mastering those levels, students cannot effectively test. However, while mastering those levels is necessary for a student to move toward reflection in action, it is not sufficient. Basic software testing provides the experience and setting for natural, recurring hypothesis testing that is important for reflection in action.

At the same time, however, there are **five perceived roadblocks** to adopting software testing practices in assignments:

1. Software testing requires experience at programming, and may be something **introductory students are not ready** for until they have mastered other basic skills.
2. Instructors just do not have the time (in terms of lecture hours) to **teach a new topic** like software testing in an already overcrowded course.
3. The course staff already has its hands full assessing program correctness—it may not be feasible to **assess test cases** too.
4. To learn from this activity, students need **frequent, concrete feedback** on how to improve their performance at many points throughout their development of a solution, rather than just once at the end of an assignment. The resources for rapid, thorough feedback at multiple points during program writing just are not available in most courses.
5. Students must **value** any practices we require alongside programming activities. A student must see any extra work as helpful in completing working programs, rather than a hindrance imposed at the instructor's desire, if we wish for students to continue using a technique faithfully.

By combining a suitable testing technique with the right assessment strategy, and supporting them with the right tools, including an automated assessment engine, it is possible to overcome all five of these difficulties.

3. TEST-DRIVEN DEVELOPMENT

To include software testing in student assignments, one must first choose a testing approach in which students will be instructed. Unfortunately, students are likely to view the software testing methods in most student-oriented software engineering texts as something that professional programmers do "out in the real world" but that has little bearing on—and provides little benefit for—the day-to-day tasks required of a student. In this case, the practice of *test-driven development* (TDD) is a better pedagogical match. TDD has been popularized by extreme programming [2]. TDD is a practical, concrete technique that students can practice on their own assignments. In TDD, one always writes one or more test cases before adding new code. The test cases capture what behavior you are attempting to produce. Then, as you write new code, these tests tell you when you have achieved your latest (small) goal.

TDD is attractive for use in an educational setting for many reasons. It is easier for students to understand and relate to than more traditional testing approaches. It promotes incremental development, promotes the concept of always having a "running version" of the program at hand, and promotes early detection of errors introduced by coding changes. It directly combats the "big bang" integration problems that many students see when they begin to write larger programs, where testing is saved until all the code writing is complete. It dramatically increases a student's confidence in the portion of the code they have finished, and allows them to make changes and additions with greater confidence because of continuous regression testing. It increases the stu-

dent's understanding of the assignment requirements, by forcing them to explore the gray areas in order to completely test their own solution. It also provides a lively sense of progress, because the student is always clearly aware of the growing size of their test suite and how much of the required behavior has already been completed. Most importantly, students begin to see these benefits for themselves after using TDD on just a few assignments.

The tool support that is available for TDD is also important. TDD frameworks are readily available, including JUnit [10] for Java, and related XUnit frameworks for other languages. Although these frameworks are aimed at professional developers, similar educational tool support is also becoming available. For example, DrJava [1], which is designed specifically as a pedagogical tool for teaching introductory programming, provides built-in support to help students write JUnit-style test cases for the classes they write. Similarly, BlueJ [11], another introductory Java environment designed specifically for teaching CS1, also provides support for JUnit-style tests. BlueJ allows students to interactively instantiate objects directly in the environment without requiring a separate main program to be written. Messages can be sent to such objects using pop-up menus. BlueJ's JUnit support allows students to "record" simple object creation and interaction sequences as JUnit-style test cases. Such tools make it easy for students to write tests from the beginning, and also mesh nicely with an objects-first pedagogy.

4. AUTOMATED GRADING

Providing appropriate feedback and assessment of student performance is critical. Many educators have used automated systems to assess and provide rapid feedback on large volumes of student programming assignments, but past approaches focus on the traditional view of program assessment—does the student submission "produce the correct output." Such a system has been in use at Virginia Tech for many years with success. Unfortunately, such tools often do little to address the issues raised here. Instead, students *focus on output correctness* first and foremost; all other considerations are a distant second at best (design, commenting, appropriate use of abstraction, testing one's own code, etc.). This is due to the fact that the most immediate feedback students receive is on output correctness, and also that students are given a clear message (say, from a zero score) when submissions do not compile, do not produce output, or do not terminate. In addition, students are not encouraged or rewarded for performing testing on their own. In practice, students do less testing on their own, often relying solely on instructor-provided sample data and the automated grading system.

In order to make classroom use of TDD practical, the challenges faced by existing automated grading systems must be addressed. Web-CAT, the Web-based Center for Automated Testing, is a new prototype tool developed at Virginia Tech for this purpose. The Web-CAT Grader grades student code and student tests together, requiring both to be present on every submission [9]. It places the burden of demonstrating correctness on the student, and then uses an assessment formula that focuses on testing performance. The Web-CAT Grader assigns scores using three measures: a score of code correctness, a score of test completeness with respect to the code, and a score of test completeness and validity with respect to the problem.

First, the *code correctness score* measures how "correct" the student's code is. To empower students in their own testing capabilities, this score is based solely on how many of the student's own tests the submitted code can pass. No separate test data from the instructor or teaching assistant is used in this score.

Second, the *test completeness score with respect to the code* measures how thoroughly the student's tests cover the student's code. For Java code, the Web-CAT Grader uses Clover [7] to instrument the student code. Coverage data is collected as student tests are run. The instructor has the option of using method coverage, statement coverage, branch coverage, or some mathematical combination to derive a measure of how thoroughly the student's code has been exercised by the student's tests.

Third, the *test completeness and validity score with respect to the problem* measures how thoroughly the student's tests cover the behavior required in the assignment. Mechanically, this is similar to a more traditional program assessment—an instructor-provided reference test suite that captures all essential behaviors is run against the student program. However, if the student program passes all the student tests, and the student tests provide reasonable coverage of the student code, then the only reason any of the reference tests can fail is because either (a) the corresponding behavior is not implemented, and thus not tested for by the student, or (b) one or more of the student-provided tests are inconsistent with the behavior required in the assignment.

All three of these measures are taken on a 0%–100% scale, and then multiplied together to produce a single composite score. As a result, the score in each dimension becomes a "cap"—it is not possible for a student to do poorly in one dimension but do well overall. Also, a student cannot accept so-so scores across the board. Instead, near-perfect performance in at least two dimensions becomes the expected norm.

To support the rapid cycling between writing individual tests and adding small pieces of code that is characteristic of TDD, the Web-CAT Grader allows *unlimited* submissions from students up until the assignment deadline. Students can get feedback any time, as often as they wish. However, their program correctness is only assessed by the tests they have written, so to find out more about errors in their own programs, a student must write the corresponding test cases. Currently, the Web-CAT Grader also applies Checkstyle and PMD, two industrial-quality static analysis tools, to assess how well the student has conformed to expected coding conventions, and all such feedback is produced in one seamless source code markup report viewable by the student on the web.

5. EXPERIENCES IN A JUNIOR COURSE

This approach has been piloted using an early version of Web-CAT in CS 3304: "Comparative Languages," a typical junior-level programming languages course at Virginia Tech. Students in the course normally write four program assignments, each requiring two to three weeks to complete. Basic instruction in TDD was provided to students, consisting of about one lecture hour of course time and several reading assignments outside of class.

In spring 2003, 59 students in the course used Web-CAT to submit all programming assignments. These students were given the same assignments used during the Spring 2001 offering of the course, where a conventional output-correctness-based automated grading system was used without TDD (students were still in-

Table 1: Score comparisons between both groups (bold differences are significant).

Comparison	Spring 2001 Without TDD	Spring 2003 With TDD	t-score Assuming Unequal Variances	Critical t-value p = 0.05
Recorded grades	**90.2%**	**96.1%**	t(*df* = 62) = **2.67**	2.00
TA assessment	98.1%	98.2%	t(*df* = 65) = 0.06	2.00
Curator assessment	93.9%	96.4%	t(*df* = 71) = 1.36	1.99
Web-CAT assessment	**76.8%**	**94.0%**	t(*df* = 61) = **4.98**	2.00
Time from first submission until assignment due	2.2 days	4.2 days	t(*df* = 112) = **3.15**	1.98
Test case failures from master suite (out of 1064)	**390 (36.7%)**	**265 (24.9%)**	t(*df* = 84) = **3.48**	1.99
Estimated Defects/KSLOC	**70.0**	**38.3**		

structed to test their own code before submission and given educational materials on basic testing practices). 59 students completed the course during spring 2001. Program submissions from both semesters were then available for detailed analysis. After assignments were turned in, the final submission of each student in both semesters was analyzed. This analysis was restricted to the first programming assignment due to manpower limitations.

Table 1 summarizes the results obtained when comparing the program submissions between the two groups. Because Web-CAT and the earlier grading system called the Curator use different grading approaches, the spring 2001 submissions were also submitted through Web-CAT for scoring. In spring 2001, however, students did not write test cases. Rather than using a fixed set of instructor-provided test data, the 2001 programs were graded using a test data generator provided by the instructor. This generator produced a random set of 40 test cases for each submission, providing broad coverage of the entire problem. To re-score each 2001 submission using Web-CAT, the generator-produced test cases originally produced for grading that submission in 2001 were submitted as if they were produced by the student.

In Table 1, "Recorded grades" represents the average final assignment score recorded in the instructor's grade book. Half of each score came from the automated assessment and half from an independent review of the student's source code by a graduate teaching assistant. "TA assessment" reflects the average amount of credit received for the TA portion of the student's grade. "Curator assessment" reflects the average amount of credit given by the traditional automated grading approach, while the "Web-CAT assessment" is the amount of credit given by the new automated assessment prototype tool.

While the "Curator assessment" average for 2003 students is slightly higher than that for 2001 students, the difference is not statistically significant. One possible interpretation for this situation is that, if any difference exists between the code produced by the two groups, the assessment approach used in 2001 was not sensitive enough to detect it. The "Web-CAT assessment" differences are significant, however. This result is understandable, since students in 2003 were given explicit feedback about how thoroughly they were testing all aspects of the problem specification, and thus had an opportunity to maximize the completeness of their tests to the best of their ability.

Finally, the student programs were analyzed to uncover the bugs they contained. One of the most common ways to measure bugs is to assess defect density, that is, the average number of defects (or bugs) contained in every 1000 non-commented source lines of code (KSLOC). On large projects, defect density data can often be collected by analyzing bug tracking databases. For student programs, however, measuring defects can be more difficult.

To provide a uniform treatment in this experiment, a comprehensive test suite was developed for analysis purposes. A suite that provided 100% condition/decision coverage on the instructor's reference implementation was the starting point. Then all test suites submitted by 2003 students and all randomly generated suites used to grade 2001 submissions were inspected, and all non-duplicating test cases from this collection were added to the comprehensive suite. For this experiment, two test cases are "duplicating" if each program in each of the student groups produces the same result (pass or fail) on both test cases. Non-duplicating test cases are thus "independent" for at least one program under consideration, but may provide redundant coverage for others. Once the comprehensive test suite was constructed, every program under consideration was run against it.

While the resulting numbers capture the relative number of defects in programs, they do not represent defect density. To get defect density information, a selection of 18 programs were selected, 9 from each group. These programs had all comments and blank lines stripped from them. They were then debugged by hand, making the minimal changes necessary to achieve a 100% pass rate on the comprehensive test suite. The total number of lines added, changed, or removed, normalized by the program length, was then used as the defects per KSLOC measure for that program. A linear regression was performed to look for a relationship between the defects/KSLOC numbers and the raw number of test cases failed from the comprehensive test suite in this sample population. This produced a correlation significant at the 0.05 level, which was then used to estimate the defects/KSLOC for the remaining programs in the two student groups.

Table 1 summarizes the results of this analysis, which show that students who used TDD and Web-CAT submitted programs containing approximately **45% fewer defects** per 1000 lines of code. While the defects/KSLOC rates shown here are far above industrial values, with values often cited around 4 or 5 defects/KSLOC, this is to be expected for student-quality code developed with no process control and no independent testing.

While the results summarized in Table 1 indicate that students do produce higher quality code using this approach, it is also important to consider how students react to TDD and Web-CAT. The 2003 students completed an anonymous survey designed to elicit their perceptions of both the process and the prototype tool. All students in the spring 2003 semester had used an automated grad-

ing/submission system before (the Curator). Students expressed a strong preference for Web-CAT over their past experiences. They found that Web-CAT was more helpful at detecting errors in their programs than the Curator (89.8% agree or strongly agree). In addition, they believed it provided excellent support for TDD (83.7% agree or strongly agree).

Students also expressed a strong preference for the benefits provided by TDD. Using TDD increases the confidence that students have in the correctness of their code (65.3% agree or strongly agree). Using TDD also increases the confidence that students have when making changes to their code (67.3% agree or strongly agree). Finally, most students **would like to use** Web-CAT and TDD for program assignments in future classes, **even if it were not required** for that course (73.5% agree or strongly agree).

6. EXPERIENCES IN CS1

As a result of experiences with this approach at the junior level, it is now being integrated into Virginia Tech's core curriculum. The fall 2003 semester began with incoming freshmen in CS1 writing basic tests of their own code in the very first laboratory session during the first week of classes. CS1 is taught in Java using BlueJ. Students are taught using an aggressive objects-first pedagogy, and begin with a variation of Bergin's Karel J. Robot simulator [3] for initial assignments. Bergin's implementation allows students to write pure Java programs using a provided Karel class library, and also provides support for JUnit-style testing. With minimal introduction to testing concepts, students readily use BlueJ to interactively instantiate objects, and then interactively "record" sequences of actions—and assertions about expected outcomes—as test cases. Finally, the Web-CAT Grader supports BlueJ's assignment submission abilities, so a student can send an assignment to the grading system just using a menu entry in their IDE, with the results popping up in their web browser.

To date, the experience has been quite positive. Allowing unlimited submissions, with a web-viewable, color-highlighted feedback report available in less than a minute, encourages frequent use by students. Further, students readily grasp the up-front emphasis that the assessment strategy gives to testing, and their natural pursuit of higher scores reinforces the desired skills. The simplicity of the tools does make this accessible, even at the CS1 level, and with minimal class time devoted to teaching testing concepts. The natural benefits that students see, together with the assessment approach, drives their use of the technique.

7. CONCLUSION

Despite the best efforts of computer science educators, CS students often do not acquire the desired analytical thinking skills that they need to be successful until later than we would like, if at all. It is possible to infuse continual practice and development of comprehension, analysis, and hypothesis-testing skills across the programming assignments in a typical CS curriculum using TDD activities. Using automated grading and feedback generation to provide for frequent, quick-turnaround assessments of student performance helps to encourage and reinforce desired behaviors. Furthermore, students see real benefits from using this approach, an important factor for its continued use across multiple courses.

Preliminary experience with TDD in the classroom and with automated assessment is very positive, indicating a significant potential for increasing the quality of student code. We plan to assess the outcomes of apply this technique in our introductory programming sequence to better characterize its impact.

8. ACKNOWLEDGMENTS

This work is supported in part by the National Science Foundation under grant DUE-0127225, and by a research fellowship from Virginia Tech's Institute for Distance and Distributed Education. Any opinions, conclusions or recommendations expressed in this paper are those of the authors and do not necessarily reflect the views of NSF or IDDL.

9. REFERENCES

[1] Allen, E., Cartwright, R., and Stoler, B. DrJava: a lightweight pedagogic environment for Java. In *Proc. 33rd SIGCSE Technical Symp. Computer Science Education*, ACM, 2002, pp. 137-141.

[2] Beck, K. *Test-Driven Development: By Example*. Addison-Wesley, Boston, MA. 2003.

[3] Bergin, J., Stehlik, M., Roberts, J., Pattis, R. Karel J. Robot: A Gentle Introduction to the Art of Object-Oriented Programming in Java. Unpublished manuscript available at: <http://csis.pace.edu/~bergin/KarelJava2ed/>

[4] Buck, D., and Stucki, D.J. Design early considered harmful: graduated exposure to complexity and structure based on levels of cognitive development. In *Proc. 31st SIGCSE Technical Symp. Computer Science Education*, ACM, 2000, pp. 75-79.

[5] Buck, D., and Stucki, D.J. JKarelRobot: a case study in supporting levels of cognitive development in the computer science curriculum. In *Proc. 32nd SIGCSE Technical Symp. Computer Science Education*, ACM, 2001, pp. 16-20.

[6] Comer, J., and Roggio, R. Teaching a Java-based CS1 course in an academically-diverse environment. In *Proc. 33rd SIGCSE Technical Symp. Computer Science Education*, ACM, 2002, pp. 142-146.

[7] Cortex, Inc. Clover: a code coverage tool for Java. Web page accessed Mar. 21, 2003: <http://www.thecortex.net/clover/>

[8] Decker, R. and Hirshfield, S. The top 10 reasons why object-oriented programming can't be taught in CS 1. In *Proc. 25th Annual SIGCSE Symp. Computer Science Education*, ACM, 1994, pp. 51-55.

[9] Edwards, S.H. Rethinking computer science education from a test-first perspective. In *Addendum to the 2003 Proc. Conf. Object-oriented Programming, Systems, Languages, and Applications*, ACM, to appear.

[10] JUnit Home Page. Web page last accessed Mar. 21, 2003: <http://www.junit.org/>

[11] Kölling, M. BlueJ—The Interactive Java Environment. Web page, last accessed Mar. 21, 2003: <http://www.bluej.org/>

[12] Krause, K.L. Computer science in the Air Force Academy core curriculum. In *Proc.13th SIGCSE Technical Symp. Computer Science Education*, ACM, 1982, pp. 144-146.

[13] Schön, D. *The Reflecting Practitioner: How Professionals Think in Action*. London: Temple Smith, 1983.

Contributing to Success in an Introductory Computer Science Course: A Study of Twelve Factors

Brenda Cantwell Wilson
Department of Computer Science
Murray State University
Murray, Ky 42071
brenda.wilson@murraystate.edu

Sharon Shrock
Department of Curriculum & Instruction
Southern Illinois University
Carbondale, IL 62901
sashrock@siu.edu

Abstract

This study was conducted to determine factors that promote success in an introductory college computer science course. The model included twelve possible predictive factors including math background, attribution for success/failure (luck, effort, difficulty of task, and ability), domain specific self-efficacy, encouragement, comfort level in the course, work style preference, previous programming experience, previous non-programming computer experience, and gender. Subjects included 105 students enrolled in a CS1 introductory computer science course at a midwestern university. The study revealed three predictive factors in the following order of importance: comfort level, math, and attribution to luck for success/failure. Comfort level and math background were found to have a positive influence on success, whereas attribution to luck had a negative influence. The study also revealed by considering the different types of previous computer experiences (including formal programming class, self-initiated programming, internet use, game playing, and productivity software use) that both a formal class in programming and game playing were predictive of success. Formal training had a positive influence and games a negative influence on class grade.

1 Introduction

Numerous studies of success in computer science including various previous computing experiences as possible predictors [2, 3, 7, 11, 12] have been conducted. Factors such as work style preference [4] and self-efficacy [1, 9, 10] as predictors have also been studied. Although attribution theory has been included in studies of other disciplines [3, 5, 6, 8], few studies have used the theory as a basis for explaining success in computer science. Attribution theory involves explanations that people give for their successes and failures. The explanations can be of a stable nature (attributing outcome to ability or difficulty of task) or an unstable nature (attributing outcome to luck or effort). The theory suggests that when people attribute their successes to unstable causes (luck or effort) and their failures to stable causes (ability or task difficulty), the probability of persistence is low.

This study was different; it included all of these factors plus other factors such as encouragement to study computer science and comfort level in the computer science course in an effort to explain success. Success was operationalized as midterm course grade. (Because of the high attrition rates in introductory computer science courses and because of the desire to study this phenomenon as it relates to the factors contributing to success in the introductory computer science course, midterm grades were used to determine success in the course to enable the inclusion of the students who drop out of the course before the end of the semester.)

2 Methodology

The study attempted to determine what relationship exists between the factors of previous programming experience, previous non-programming experience, attribution for success/failure, self-efficacy, comfort level, encouragement from others, work style preference, math background, and gender of introductory computer programming students and their midterm course grade. Also, the study sought to discover which of the above mentioned factors are predictive of midterm course grade. Also, a look at different types of previous computing experiences to determine whether they are predictive of success in CS1 was included.

2.1 Subjects

Approximately 130 students were enrolled in six sections of CS 202 Introduction to Computer Science at a comprehensive Midwestern university (approximately 22,000 student population) during the spring of 2000.

SIGCSE 2001 2/01 Charlotte, NC, USA
 ISBN 1-58113-329-4/01/0002...$5.00

There were 105 students who voluntarily participated in the study. CS 202 is the first programming class required in the computer science major and uses C++ as the programming language.

2.2 Instruments

Two instruments were used to collect data from the subjects: a questionnaire and the Computer Programming Self-Efficacy Scale developed by Ramalingam & Wiedenbeck [10]. The questionnaire collected data on the following items: gender, math background (number of semesters of high school math classes taken), previous programming experiences, previous non-programming computer experiences, encouragement by others to pursue computer science as a career, comfort level, work style preference, and attribution for perceived "success" or "failure" on the midterm exam. A pilot test was given to enable the researcher to find any ambiguities in the questionnaire, and revisions were made appropriately. One expert in the field of psychology research and two experts in the field of testing and evaluation were asked to evaluate the face validity of the questionnaire. Four seasoned computer science professors examined the content of the instrument. The questionnaire was found to have high content validity for measuring the variables in the study. A test-retest was used to examine the reliability of the questionnaire. The instrument was administered to students in two sections of an introductory computer science course at another university. Because the questionnaire was intended to measure different attributes, it was necessary to determine eight correlations. The Pearson Correlation coefficients were .98 for math background, 1.0 for previous programming course, .72 for previous self-initiated programming experience, .95 for previous non-programming experience, .80 for work style preference, .88 for comfort level, .72 for attributions to success/failure, and 1.0 for encouragement. The Computer Programming Self-Efficacy Scale was used to collect data on domain-specific self-efficacy as it relates to tasks in the C++ programming language. The authors reported an overall alpha reliability of .98 on the instrument.

2.3 Predictor Variables

Twelve predictor variables were included in the study. The way in which each variable was measured is described below: (all of this data except for self-efficacy, and midterm course grade were collected via the questionnaire)

1. Gender – a dichotomous variable (male or female).

2. Previous programming experience – a dichotomous variable determined by whether the subject had engaged in any programming prior to the course.

   In order to study the types of previous programming experience, this variable was subdivided into two areas: (a) formal programming course taken – a dichotomous variable determined by whether the subject had a previous programming course or not and (b) self-initiated programming experience – a dichotomous variable determined by whether the subject learned to write programs separate and apart from a formal class.

3. Previous non-programming experience – a dichotomous variable determined by whether the subject had engaged in non-programming computer activities.

   In order to study the types of previous non-programming experience, this variable was subdivided into three areas: (a) internet experience – a continuous variable determined by the number of hours per week each subject reported using the internet, e-mail, chat-rooms, and/or on-line discussion groups, (b) games – a continuous variable determined by the number of hours per week the subject reported spending time playing games on the computer, and (c) use of productivity software – a continuous variable determined by the number of hours per week the subject reported using productivity software such as word processing, spreadsheets, databases, and/or presentation software packages.

4. Encouragement to pursue computer science – a dichotomous variable representing whether the subject received encouragement to pursue computer science or not.

5. Comfort level – a continuous variable derived from seven questions on the questionnaire regarding asking and answering questions in class, in lab, and during office hours; anxiety level while working on computer assignments; perceived difficulty of course; perceived understanding of concepts in the course as compared with classmates; and perceived difficulty of completing the programming assignments. Numbers were appropriately assigned to the choices of the BAR (Behaviorally Anchored Rating) scale in these questions and summed for a composite score of comfort level.

6. Work style preference– a dichotomous variable (competitive or cooperative) representing the answer to a question about preference for writing computer programs and/or studying for exams.

7. Attributions – four continuous variables derived from the subject's rating of possible reasons for success or failure on the midterm exam. They are: (a) attribution to ability, (b) attribution to task ease/difficulty, (c) attribution to luck, and (d) attribution to effort.

8. Self-efficacy – a continuous variable, which is the summation of the choices made on a Likert-type scale from the Computer Programming Self-Efficacy Scale.

9. Math background – a continuous variable represented by the number of semesters of high school math courses reported by the subject.

2.4 Criterion Variable

The criterion variable of the study was the midterm grade in the introductory computer science class for each student. This was a continuous variable representing a number between 0 and 100.

To ascertain that the use of the midterm grade was a viable choice for determining success in the computer programming class, a correlation coefficient was generated using the midterm scores and the final scores in two sections of the first course in Computer Science from the fall semester of 1999. The Pearson Correlation Coefficient was extremely high and significant, $r = .97173$, $N = 48$, $p = .0001$, therefore, it seemed reasonable that the midterm grade was a good indicator of success in the class.

2.5 Procedure

During the spring semester the questionnaire and Computer Programming Self-Efficacy Scale were distributed after the exam and before midterm of the semester at a class lecture session. Data was collected from 105 students.

2.6 Analysis of Data

Although no study could be found that combined all of the predictor variables that are included in this study, some of the previous research could be used to determine an expected hierarchy of predictor variables. Therefore, based on the literature review and on the researcher's experience of teaching computer science, a hierarchical model was generated and tested using the general linear model. The model included twelve predictor variables in the following order: math, previous programming experience, attribution to luck, attribution to difficulty of task, comfort level, non-programming experience, work style preference, domain-specific self-efficacy, encouragement to study computer science, attribution to effort, attribution to ability, and gender. This model was tested and compared to the findings of the previous research studies in computer science success. All analyses used an alpha level of .05 to determine significance.

A residual plot was generated from the data confirming the multi-linear model. A correlation matrix was generated to examine how each of the 12 factors correlated with midterm grade and with each of the other predictor variables. By examining the R^2 and its p-value of the full-model regression equation, the proportion of variance in midterm grade accounted for by the twelve predictor variables was determined. The Type I sums of squares and Type III sums of squares with associated p-values were examined to determine the contribution of each factor over and above the other factors. The parameter estimates from the multiple regression tests were also examined to see whether each factor had a positive or negative effect on midterm grade. The full model for the twelve predictor variables was:

$Y = a_0U + a_1X1 + a_2X2 + a_3X3 + a_4X4 + a_5X5 + a_6X6 + a_7X7 + a_8X8 + a_9X9 + a_{10}X10 + a_{11}X11 + a_{12}X12 + E$

where Y = midterm course grade
X1 = 1 if previous programming; 0 otherwise
X2 = 1 if previous non-programming; 0 otherwise
X3 = rating for attribution to task difficulty
X4 = rating for attribution to luck
X5 = rating for attribution to effort
X6 = rating for attribution to ability
X7 = rating for self-efficacy
X8 = rating for comfort level
X9 = 1 if had encouragement; 0 otherwise
X10 = 1 if male; 0 if female
X11 = number of sem. of math courses
X12 = 0 if work style preference is cooperative; 1 if competitive
E = the errors of prediction

To determine if any of the previous computing experiences were predictive of success, a full model and four restricted models were used. The restricted models were constructed by dropping out one predictor variable from the full model. Each restricted model was tested against the full model to ascertain whether the contribution of each predicting factor over and above the other factors in combination was significant. The full model for previous computing experiences was:

$Y = b_0U + b_1P1 + b_2P2 + b_3P3 + b_4P4 + b_5P5 + + E$

where Y = midterm course grade
P1 = 1 if programming class; 0 otherwise
P2 = 1 if self-initiated programming; 0 otherwise
P3 = number of hours/week of Internet use
P4 = number of hours/week of games
P5 = number of hours/week of productivity software use
E = errors of prediction

3 Results

The proportion of variance in midterm score accounted for by the linear combination of the 12 factors was approximately .44, $R^2 = .4443$, which was statistically significant, $F(12, 92) = 6.13$, $p = .0001$. Three of the predictor variables contributed a significant difference in the midterm grade at the .05 level even after being considered last in the model. They were comfort level, math background, and attribution of success/failure to luck with p-values of .0002, .0050, and .0233 respectively. Two of the three significant predictive factors (comfort level and math) had positive correlations with the midterm score, but attribution of success/failure to luck had negative parameter estimation. (See Table 1.)

When stepwise multiple regression was used, two more variables showed significant influence in a five factor model. They were work style preference and attribution of success/failure to task difficulty. These five variables contributed to 40% of the variance. The work style preference was positively correlated to the midterm score, which indicated that an individual/competitive work style preference had a positive influence on the midterm score. Attribution to task difficulty was negatively correlated to midterm score.

Two of the previous computing experience variables showed significant influence in predicting the midterm score: previous programming course and games with p-values of .0006 and .0287 respectively. (See Table 2.) It was also noted that while the previous programming course variable had a positive influence on midterm grade, games

had a negative influence. Also the proportion of variance accounted for by the five previous programming and non-programming variables was .15 which was significant for the sample, p = .0041.

Table 1: Summary of Type I and Type III Sums of Squares General Linear Models Procedure
(R^2 = .444347, F = 6.13, P = .0001)

Source	DF	Type I SS	F	P
Math	1	2348.67	15.17	.0002
Prg	1	1203.42	7.77	.0064
Luck	1	2014.73	13.02	.0005
Diff	1	766.75	4.95	.0285
Cmf-lev	1	3460.21	22.36	.0001
NPrg	1	51.19	0.33	.5666
WPrf	1	625.34	4.04	.0474
SE	1	53.60	0.35	.5577
Enc	1	302.55	1.95	.1654
Effort	1	359.15	2.32	.1311
Ability	1	2.20	0.01	.9053
Gender	1	199.63	1.29	.2590

Source	DF	Type III SS	F	P
Math	1	1279.79	8.27	.0050
Prg	1	389.52	2.52	.1161
Luck	1	824.10	5.32	.0233
Diff	1	455.95	2.95	.0895
Cmf-lev	1	2334.38	15.08	.0002
NPrg	1	169.47	1.09	.2981
WkPrf	1	482.72	3.12	.0807
SE	1	12.53	0.08	.7767
Enc	1	170.09	1.10	.2973
Effort	1	296.74	1.92	.1695
Ability	1	3.42	0.02	.8821
Gender	1	199.63	1.29	.2590

4 Conclusion

Comfort level in the computer science class was the best predictor of success in the course. Math background was second in importance in predicting success in this computer science class. It is most interesting, in this study, that comfort level was found to be more important than math background. Most of the research studied for the literature review, which included math as a predictor, concluded that math and computer programming experience were the most important factors in success in computer science, although many of these studies did not include studying comfort level as such. Although programming experience (which included both a previous programming course and self-initiated programming) was not found to be significant in the full model, when the different types of computing experiences were compared as predictors of midterm grade, the previous programming course and game playing were both significant. It should be noted that the notion that game playing gives students an "edge" in a computer science course was not supported in this study. Game playing had a negative effect on the midterm grade.

The result for analysis of attribution to luck was also an interesting finding. To support most of the attribution research findings, attribution to luck would only be positively correlated to success in the course for those students who were unhappy with their score. In other words, if they could attribute their "low" score to an unstable cause such as luck, then they would continue to try to do better. In this study, however, attribution to luck for all students (whether happy or unhappy with their scores) was negatively correlated to midterm.

Table 2: Summary of Multiple Regression Analysis on Previous Programming and Non-Programming Experience

Analysis of Variance for Model				
Source	Df	R^2	F value	P
Model	5	.1577	3.706	.0041
Error	99			
C Total	104			
Parameter Estimates				
Variable	Parameter estimate	Standard error	T	P
Intercept	64.8755	2.6049	24.905	.0001
PrvPrgCs	10.6138	2.9944	3.545	.0006
SiPrg	1.8392	3.3532	.548	.5846
Int	.0906	.1765	.514	.6087
Games	-.4217	.1900	-2.219	.0287
Apps	.2315	.1761	1.315	.1917

Note. Variables
PrvPrgCs = previous programming course; SiPrg = self-initiated programming; Int = use of the Internet; Games = playing games on the computer; Apps = use of productivity software

5 Recommendations

Although this study did not show that higher comfort levels "cause" students to perform better in the computer science class, because of the positive correlation in this study between comfort level and success in the introductory computer science course, the notion that providing the optimum class environment for producing higher levels of comfort for students is at least warranted. It is suggested that professors of college computer science should understand the importance of providing an environment in the course which encourages students to ask and answer questions, both in class and outside of class, in a way that allows the students to feel comfortable and not intimidated. Opportunities for students to be able to consult with faculty, teaching assistants, or tutors were also indicated. The recent move in many universities to force students into large lecture sections for computer science, which by its very nature discourages dialogue between students and faculty, is an indication of the misunderstanding of the importance of the level of comfort students may need in this difficult discipline. Also, advisers should stress an appropriate mathematical background for students wanting

to pursue computer science. Finally, since attribution to luck showed a negative correlation with success, professors should endeavor to match class assignments and exam questions in the hope that students will not perceive luck as a reason for success or failure on the exams. Again, this suggestion is warranted even though the study only showed a negative correlation and not causation.

References

[1] Bandura, A. Self-efficacy: Toward a unifying theory of behavioral change. *Psychological Review, 84(2)*, (1977), 191-215.

[2] Bunderson, E.D., & Christensen, M.E.. An analysis of retention problems for female students in university computer science programs. *Journal of Research on Computing in Education 28(1)*, (1995), 1-15.

[3] Clarke, V.A., & Chambers, S.M. Gender-based factors in computing enrollments and achievement: Evidence from a study of tertiary students. *Journal of Educational Computing Research, 5(4)*, (1989), 409-429.

[4] Cottrell, J. I'm a stranger here myself: A consideration of women in computing. *Proceedings of the 1992 ACM SIGUCCS User Services Conference*, (1992), 71-76.

[5] Deboer, G.E. Factors related to the decision of men and women to continue taking science courses in college. *Journal of Research in Science Teaching, 21(3)*, (1984), 325-329.

[6] Dweck, C.S., & Leggett, E.L. A social-cognitive approach to motivation and personality. *Psychological Review (95)*, (1988), 256-273.

[7] Kersteen, Z.A., Linn, M.C., Clancey, M., & Hardyck, C. Previous experience and the learning of computer programming: The computer helps those who help themselves. *Journal of Educational Computing Research 4(3)*, (1988).

[8] Maeher, M.L. On doing well in science: Why Johnny no longer excels; why Sarah never did. In S.G. Paris, G.M. Olsen, & H.W. Stevenson (Eds.), *Learning and Motivation in the Classroom* (Chapter 8), (1988).

[9] Miura, I.T. The relationship of computer self-efficacy expectations to computer interest and course enrollment in college. *Sex Roles, 16(5)*, (1987), 303-311.

[10] Ramalingam, V., & Wiedenbeck, S. Development and validation of scores on a computer programming self-efficacy scale and group analyses of novice programmer self-efficacy. *Journal of Educational Computing Research, 19(4)*, (1998), 367-381.

[11] Taylor, H., & Mounfield, L. An analysis of success factors in college computer science: High school methodology is a key element. *Journal of Research on Computing in Education, 24(2)*, (1991), 240-245.

[12] Taylor, H., & Mounfield, L. Exploration of the relationship between prior computing experience and gender on success in college computer science. *Journal of Educational Computing Research 11(4)*, (1994), 291-306.

Teaching Objects-first In Introductory Computer Science

Stephen Cooper*
Computer Science Dept.
Saint Joseph's University
Philadelphia, PA 19131
scooper@sju.edu

Wanda Dann*
Computer Science Dept.
Ithaca College
Ithaca, NY 14850
wpdann@ithaca.edu

Randy Pausch
Computer Science Dept.
Carnegie Mellon University
Pittsburgh, PA 15213
pausch@cmu.edu

Abstract
An objects-first strategy for teaching introductory computer science courses is receiving increased attention from CS educators. In this paper, we discuss the challenge of the objects-first strategy and present a new approach that attempts to meet this challenge. The new approach is centered on the visualization of objects and their behaviors using a 3D animation environment. Statistical data as well as informal observations are summarized to show evidence of student performance as a result of this approach. A comparison is made of the pedagogical aspects of this new approach with that of other relevant work.

Categories and Subject Descriptors
K.3 [**Computers & Education**]: Computer & Information Science Education – *Computer Science Education.*

General Terms
Documentation, Design, Human Factors,

Keywords
Visualization, Animation, 3D, Objects-First, Pedagogy, CS1

1 Introduction
The ACM Computing Curricula 2001 (CC2001) report [8] summarized four approaches to teaching introductory computer science and recognized that the "programming-first" approach is the most widely used approach in North America. The report describes three implementation strategies for achieving a programming-first approach: imperative-first, functional-first, and objects-first. While the first two strategies have been utilized for quite some time, it is the objects-first strategy that is presently attracting much interest. Objects-first "emphasizes the principles of object-oriented programming and design from the very beginning…. [The strategy] begins immediately with the notions of objects and inheritance….[and] then goes on to introduce more traditional control structures, but always in the context of an overarching focus on object-oriented design" [8, Chapter 7].

*This work was partially supported by NSF grant DUE-0126833

SIGCSE'03 February 19-23, 2003, Reno, Nevada, USA.

The Challenge of Objects-first: The authors of CC2001 admit that an objects-first strategy adds complexity to teaching and learning introductory programming. Why is this so? The classic instruction methodology for an introduction to programming is to start with simple programs and gradually advance to complex programming examples and projects. The classic approach allows a somewhat gentle learning curve, providing time for the learner to assimilate and build knowledge incrementally. An objects-first strategy is intended to have students work immediately with objects. This means students must dive right into classes and objects, their encapsulation (public and private data, etc.) and methods (the constructors, accessors, modifiers, helpers, etc.). All this is in addition to mastering the usual concepts of types, variables, values, and references, as well as with the often-frustrating details of syntax. Now, add event-driven concepts to support interactivity with GUIs! As argued by [11], learning to program objects-first requires students grasp "many different concepts, ideas, and skills…almost concurrently. Each of these skills presents a different mental challenge."

The additional complexity of an objects-first strategy is understood when considered in terms of the essential concepts to be mastered. The functional-first strategy initially focuses on functions, deferring a discussion of state until later. The imperative-first strategy initially focuses on state, deferring a discussion of functions until later. The objects-first strategy requires an initial discussion of both state and functions. The challenge of an objects-first strategy is to provide a way to help novice programmers master both of these concepts at once.

2 Instructional Support Materials
In response to interest in an objects-first approach, several texts and software tools have been published/developed that promote this strategy (such as [1, 12]). Four recent software tools are worthy of mention as using an objects-first approach: BlueJ [9], Java Power Tools [11], Karel J. Robot [2], and various graphics libraries. Interestingly, all these tools have a strong visual/graphical component; to help the novice "see" what an object actually is – to develop good intuitions about objects/object-oriented programming.

BlueJ [9] provides an integrated environment in which the user generally starts with a previously defined set of classes. The project structure is presented graphically, in UML-like fashion. The user can create objects and invoke methods on those objects to illustrate their behavior. Java Power Tools (JPT) [11] provides a comprehensive, interactive GUI, consisting of several classes with which the student will work. Students interact with the GUI, and learn about the behaviors of the GUI classes through this interaction. Karel J. Robot [2] uses a microworld with a robot to help students learn about objects. As in Karel [10], Robots are

added to a 2-D grid. Methods may be invoked on the robots to move and turn them, and to have the robots handle beepers. Bruce et al. [3] and Roberts [13] use graphics libraries in an object-first approach. Here, there is some sort of canvas onto which objects (e.g. 2-D shapes) are drawn. These objects may have methods invoked on them and they react accordingly.

In the remainder of this paper, we present a new tactic and software support for an objects-first strategy. The software support for this new approach is a 3D animation tool. 3D animation assists in providing stronger object visualization and a flexible, meaningful context for helping students to "see" object-oriented concepts. (A more detailed comparison of the above tools with our approach is provided in a later section.)

3 Our Approach

Our motivation in researching and developing this new approach is to meet the challenge of an objects-first approach. Our approach meets the challenge by:

- Reducing the complexity of details that the novice programmer must overcome
- Providing a design first approach to objects
- Visualizing objects in a meaningful context

In this approach, we use Alice, a 3D interactive, animation, programming environment for building virtual worlds, designed for novices. The Alice system, developed by a research group at Carnegie Mellon under direction of one of the authors, is freely available at www.alice.org. A brief description of the interface is provided.

Figure 1. The Alice Interface

Alice provides an environment where students can use/modify 3D objects and write programs to generate animations. A screen-capture of the interface is shown in Figure 1. The interface displays an object tree (upper left) of the objects in the current world, the initial scene (upper center), a list of events in this world (upper right), and a code editor (lower right). The overlapping window tabs in the lower left allow for querying of properties, dragging instructions into the code editor, and the use of sound.

Student Programs: A student adds 3D objects to a small virtual world and arranges the position of each object in the world. Each object encapsulates its own data (its private properties such as height, width, and location) and has its own member methods. While it is beyond the scope of this paper to discuss all the details, a brief example is discussed below to illustrate some of the principles. Interested readers may wish to read [4, 6, 7] for a more complete description. Figure 2 contains an initial scene that includes a frog (named kermit), a beetle (ladybug), a flower (redFlower), and several other objects around a pond.

Figure 2. An initial scene in an Alice world

Once the virtual world is initialized, the program code is created using a drag-and-drop smart editor. Using the mouse, an object is mouse-clicked and dragged into the editor where drop-down menus allow the student to select from primitive methods that send a message to the object. A student can write his/her own user-defined methods and functions, and these are automatically added to the drop-down menus.

In this example, the task is for kermit to hop over to the ladybug. The code is illustrated in Figure 3. It is interesting to note that the built-in predicates ("Questions" in Alice-lingo) "is at least m meters away from n", "is within x meters of y", and "is in front of z" all return spacial information about the objects in question. (Users may define their own, user-defined, questions, at both the world-level as well as at the character-level.) The *bigHop(number n)* and *littleHop()* methods are both character-level. In other words, the basic frog class has been extended to create a frog that knows how to make a small hop and how to hop over a large object (receiving a parameter as to how high it must hop).

This example illustrates some important aspects of our approach. The mechanism for generating code relies on visual formatting rather than details of punctuation. The gain from this no-type editing mechanism is a reduction in complexity. Students are able to focus on the concepts of objects and encapsulation, rather than dealing with the frustration of parentheses, commas, and semicolons. We hasten to note that program structure is still part of the visual display and the semantics of instructions are still learned. A switch is used to display Java-like punctuation to support a later transition to C++/Java syntax.

Three-dimensionality provides a sense of reality for objects. In the 3D world, students may write methods from scratch to make objects perform animated tasks. The animation task provides a meaningful context for understanding classes, objects, methods, and events.

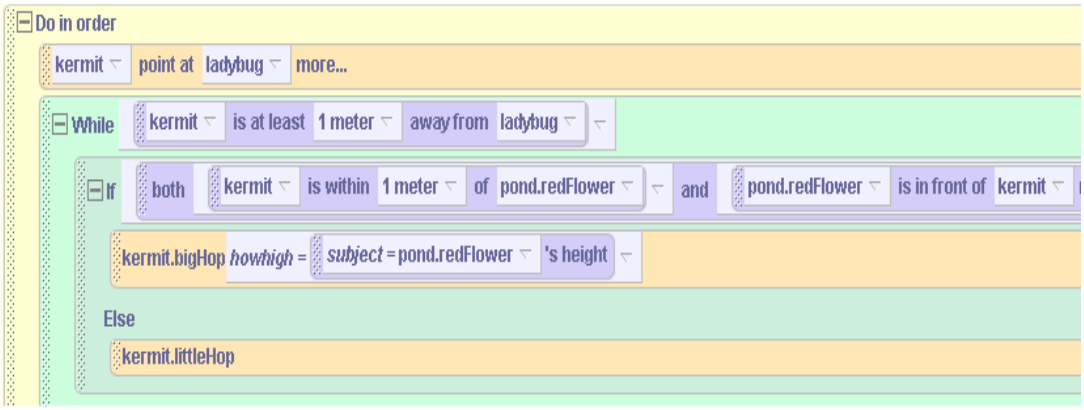


Figure 3. The code to have kermit hop over to the ladybug

4 Observations

We have been teaching and researching this new objects-first approach in an introduction to programming course for the past 3 years. One of the authors uses this approach in a ½ semester course that students take concurrently with CS1. Another author uses this approach as part of a course that students take before CS1. While early quantitative results are discussed in the next section, we present more informal observations in this section.

Strengths: We have seen that students develop:

- A strong sense of design. In our approach, we use storyboarding and pseudocode to develop designs. This may be influenced by the nature of our open-ended assignments. However, we see students in later classes writing down their thoughts about an assignment on paper first, before going to the computer.
- A contextualization for objects, classes, and object-oriented programming. We believe that this is one of the big "wins" for our approach. Everything in the student's virtual world is an object! Exercises and lab projects set up scenes where objects fly, hop, swim, and interact in highly imaginative movie-like simulations and games.
- An appreciation of trial and error. Students learn to "try out" individual animation instructions as well as their user-defined methods. Each animation instruction causes a visible change in the animation. Students learn to relate individual instructions, and methods to the animated action on the screen [7]. This direct relationship can be used to support development of debugging skills.
- An incremental construction approach, both for character (class)-level as well as world-level methods. Students do not write the whole program first. They program incrementally, one method at a time, testing out each piece.
- A firm sense of objects. The strong visual environment helps here.
- Good intuitions concerning encapsulation. Some state information can be modified by invoking methods on an object. For example, an object's position can be changed by invoking a *move* method. But the actual spatial coordinates that represent the object's position cannot be directly accessed.
- The concept of methods as a means of requesting an object to do something. The way to make an object perform a task is to send the object a message.
- A strong sense of inheritance, as students write code to create more powerful classes.
- An ability to collaborate. Students work on building the characters individually and then combine them to build virtual worlds and animations in group projects.
- An understanding of Boolean types. Students are prevented, by the smart-editor, from dragging incorrect data-type expressions into if statements and loops, for example.
- A sense of the program state. This is of particular importance, as mentioned earlier in this paper. This topic is discussed at length in [7].
- An intuitive sense of behaviors and event-driven programming.

One other observation is that it is possible to have students either create their programs from scratch or to build virtual worlds with characters which already have many specialized methods pre-defined. This latter case allows students to experiment with modifying existing classes/programs.

Weakness: A strength of our approach is also a source of weakness. Students do not develop a detailed sense of syntax, even with the C++/Java syntax switch turned on, as they only drag the statements/expressions into the code window. They do not get the opportunity to experience such errors as mismatched braces, missing semicolons, etc. Our experience with students making the transition from Alice to C++/ Java is that students quickly master the syntax.

5 Results

Table 1 illustrates the results of students at Ithaca College and Saint Joseph's University who took a course using our proposed approach during the 2001-2002 school year. The weakest 21 CS majors (defined as those CS students who were not ready for calculus and who had no previous programming experience) were invited to take a course using our approach, either concurrent with, or preliminary to CS1. 11 of the 21 students took the course,

while 10 did not. (Some students who did not take the course had scheduling conflicts.)

Statistics	All	Test	Control
# Students	49	11	10
Mean	2.49	2.8	1.3
Median	2.75	3	1.25
Variance	1.62	0.75	1.22

Table 1: Students taking Alice, 2001-2002

The results show that the 11 students who took the Alice-based course did better in CS1 than the total group, and significantly better than the 10 students who were of a similar background. Not only did the control group perform better in CS1, the lower variance indicates that a smaller percentage of those students performed poorly in CS1. Perhaps the most telling statistic is the percentage of students who continued on to CS2, the next computer science class. 65% of all the students who took CS1 continued on to CS2. Of the students in the test group (who took our course with Alice), 91% continued on to CS2. Only 10% of the control group enrolled in CS2. A larger group of students is being studied (in much more detail) this current (2002-2003) academic year, as part of an NSF supported study.

The authors have a textbook (to be published by Prentice-Hall for Fall 2003). An early draft is available at www.ithaca.edu/wpdann/alice2002/alicebook.html The URL for the solutions is available by contacting the authors. And, a set of lecture notes and sample virtual worlds is available at: http://www.sju.edu/~scooper/fall02csc1301/alice.html

6 Comparison with other tools

In this section we explore what we consider to be our relative strengths and weaknesses as compared to other object-first tools mentioned earlier. It is important to note that, as we have not seen studies detailing actual effectiveness of many of the other tools, we are hesitant to state too strongly the degree to which we think such tools do or do not work.

Events: JPT makes heavy use of GUIs, and both JPT and Bruce's ObjectDraw library rely on event-driven programming. Kölling and Rosenberg [9] state that building GUIs is "very time intensive", and argue that the GUI code is an "example that has very idiosyncratic characteristics that are not common to OO in general." Culwin [5] argues "the design of an effective GUI requires a wider range of skills than those of software implementation…. Even if an optimal interface is not sought at this stage it must be emphasized to students…that there is much more to the construction of a GUI than the collecting together of a few widgets and placing it in front of the user." While we might not go as far as these criticisms, it is clear that event handling does add a layer of complexity. In our approach, the use of events is optional and is accomplished through the use of several powerful primitives. This makes the presentation of events and event handling quite simple. We disagree with the statement "it is not possible to do Objects-first" without also doing GUI First!"[11], as both our approach and some of the graphics libraries do accomplish an object-first approach without the use of a GUI (though adding events generally makes virtual worlds much more fun for the students).

Modifying existing code: BlueJ and JPT depend on starting with programs that consist of previously written code. Bruce is concerned "these approaches will leave students feeling they have no understanding of how to write complete programs." The BlueJ and JPT authors maintain that, due to complexity of object-oriented design, it is favorable for novices to start with partially/completely developed projects and to modify them. Our approach allows the instructor to choose to use partially developed programs in introductory worlds. But, we generally have students build virtual worlds from scratch.

Use of the tool throughout the CS1 course: Each of these tools, with the exception of Karel J. Robot, is (or at least seems to be) capable of being used throughout the CS1 course. We have designed lecture materials to be used as an initial introduction to object-oriented programming, occupying the first 3-6 weeks of a CS1 course. It would be possible to intersperse the teaching of Alice with the teaching of, say, Java, throughout the semester.

Complexity of syntax: The use of graphics libraries is likely the most complex approach. Even though libraries are provided, students still must write Java/C++ programs from scratch, mastering a non-trivial amount of syntax (regardless whether they understand the semantics of what they are writing). Then they need to understand the particulars of the graphics library. Karel J. Robot has a fair bit of Java that needs to be mastered before being able to write a program. The BlueJ and JPT approaches are somewhat simpler, as students only modify existing code. Yet, it is still necessary to write correct Java code, and certain errors (such as missing brackets or trying to place code in the wrong location, or invoking a method with a bad parameter) can lead to errors in the code provided to the student -- and the student may not know how to start debugging code that he/she did not write.

Concurrency: As Culwin writes [5], "if an early introduction of GUIs is advocated within an object first approach, the importance of concurrency cannot be avoided." Alice supports concurrency, providing primitives for performing actions simultaneously.

Examples: This is a challenge for all objects-first approaches. Developing a large collection of examples (whether to be used as instructional aids, assignments or exam questions) is a time-consuming task that must be solved if these tools, together with their associated approach are to be successful. One product of our research efforts is a resource of examples, exercises, and projects with solutions. It does need to be made larger, which we are doing each semester.

7 Conclusions

The authors strongly believe that, as long as object-oriented languages are the popular language of choice in CS1, the objects-first approach is the best way to help students master the complexities of object-oriented programming. We believe that other tools mentioned here are quite useful in teaching objects-first. (We have used most of them ourselves.) We have been particularly impressed with the results we have seen so far with the approach we have presented here – we have been able to significantly reduce the attrition of our most at-risk majors. The current NSF study will examine the effectiveness of our proposed approach in greater detail, and with larger numbers of students. Additionally, we hope to gain feedback from some of the additional institutions that are using our materials and our approach.

References

[1] Arnow, D. and Weiss, G. Introduction to programming using Java: an object-oriented approach, Java 2 update. Addison-Wesley, 2001.

[2] Bergin, J., Stehlik, M., Roberts, J., and Pattis, R. *Karel J. Robot a gentle introduction to the art of object oriented programming in Java*. Unpublished manuscript, available [August 31, 2002] from: http://csis.pace.edu/~bergin/KarelJava2ed/Karel++Java Edition.html

[3] Bruce, K., Danyluk, A., & Murtagh, T. A library to support a graphics-based object-first approach to CS 1. In *Proceedings of the 32nd SIGCSE technical symposium on Computer Science Education* (Charlotte, North Carolina, February, 2001), 6-10.

[4] Cooper, S., Dann, W., & Pausch, R. Using animated 3d graphics to prepare novices for CS1. *Computer Science Education Journal*, to appear.

[5] Culwin, F. Object imperatives! In *Proceedings of the 30th SIGCSE technical symposium on Computer Science Education* (New Orleans, Louisiana, March, 1999), 31-36.

[6] Dann, W., Cooper, S., & Pausch, R. Using visualization to teach novices recursion. In *Proceedings of the 6th annual conference on Innovation and Technology in Computer Science Education* (Canterbury, England, June, 2001), 109-112.

[7] Dann, W., Cooper, S., & Pausch, R. Making the connection: programming with animated small worlds. In *Proceedings of the 5th annual conference on Innovation and Technology in Computer Science Education* (Helsinki, Finland, July, 2000), 41-44.

[8] Joint Task Force on Computing Curricula. Computing Curricula 2001 Computer Science. *Journal of Educational Resources in Computing (JERIC)*, 1 (3es), Fall 2001.

[9] Kölling, M. & Rosenberg, J., Guidelines for teaching object orientation with Java. In *Proceedings of the 6th annual conference on Innovation and Technology in Computer Science Education* (Canterbury, England, June, 2001), 33-36.

[10] Pattis, R., Roberts, J, & Stehlik, M. *Karel the robot: a gentle introduction to the art of programming, 2nd Edition*. John Wiley & Sons, 1994.

[11] Proulx, V., Raab, R., & Rasala, R. Objects from the beginning – with GUIs. In *Proceedings of the 7th annual conference on Innovation and Technology in Computer Science Education* (Århus, Denmark, June, 2002), 65-69.

[12] Riley, D. *The object of Java: Bluej edition*. Addison-Wesley, 2002.

[13] Roberts, E. & Picard, A. Designing a Java graphics library for CS1. In *Proceedings of the 3rd annual conference on Innovation and Technology in Computer Science Education* (Dublin, Ireland, July, 1998), 213-218.