

**Teaching Object-oriented Design Without Programming:
A Progress Report**

Judith E. Sims-Knight
Department of Psychology
University of Massachusetts Dartmouth

and

Richard L. Upchurch
Department of Computer and Information Science
University of Massachusetts Dartmouth

Running Head:
Teaching Object-oriented Design Without Programming

This research was supported by National Science Foundation Grant No. MDR-9154008.

Sims-Knight, J. E., & Upchurch, R. L. (1993). Teaching Object-oriented Design Without Programming: A Progress Report. Computer Science Education, 4, 135-156.

Abstract

This project is demonstrating the feasibility of using the object-oriented paradigm to teach students software design in a nonprogramming context. The program, developed using principles of user-based, prototyping design, teaches students to create responsibility-driven designs of computer games. Investigations with high school students with little or no knowledge of computers and senior computer science majors have demonstrated that students can indeed learn to use Class-Responsibility-Collaborator (CRC) cards to produce creditable high-level designs in a relatively short time whether or not they have programming experience and can generalize what they have learned to a new design. Computer science majors created more complete designs and demonstrated a deeper understanding of the design process than the high school students, but they still found the experience valuable and they still showed room for improvement. Both samples generally find the process interesting and relatively painless.

1.0 Introduction

Studies of the learning outcomes of introductory programming courses have shown relatively poor understanding of the programming language under study, little development of problem solving skills, and little progress toward the central issues of analysis and design, e. g., [1], [2], [3], [4], [5], [6], [7], [8], [9], [10]. In addition, both research and the practical experiences of expert software designers have suggested that the analysis of the problem domain and the design of an overall plan of the software are more central to the development of software than is learning to code. Students in programming courses that emphasize design learn more than students in other courses, but they still show little progress toward design skills or general problem solving skills [4], [11], [12].

Research on how designers design suggest why teaching design within the context of teaching a programming language is so difficult. This research [9], [13], [14], [15], [16], [17], [18], [19] and insights from software engineers [20], [21], [22], [23], [24], [25] give us two general characteristics of the process of designing software:

1. The process of discovery (the iterative reflection between analysis and design phases and between levels of design phases) should be structured in general by the principles of top-down decomposition and successive refinement, but there should be freedom to deviate.
2. Designers need to refocus repeatedly, systematically, and iteratively on the overall structure, that is, on the general functions of the parts and how they interrelate with each other to form the whole, and to cycle between analysis and design representations.

Both of these characteristics require thinking at a higher level than the code level. Teaching students to chunk their programming code into modules and organize those modules according to principles of flow diagrams may be an inadequate means of teaching such skills. It might be preferable to teach students design skills independent of a programming language. If feasible, this approach makes sense four ways:

1. Design is one of the three central processes in computer science, according to the Task Force on the Core of Computer Science [26] and current accreditation guidelines.
2. Teaching software design would result in a broader introduction to computer science as measured by the Task Force's recommendations. Students would still get an appreciation of how

programming accomplishes its goals when they learn about the design of a program. In addition, they would get an introduction to two more of the nine subfields identified in the report--software engineering and human computer interaction.

3. Progress in learning analysis and design should be more effective and more rapid if students can focus on those issues rather than having to master them in the context of the complex procedural skills of a programming language.
4. If one is freed from the necessity of selecting programming tasks simple enough for novice coders, one can select activities and examples that are (a) more complex, thus more realistic, and (b) more interesting, thus enabling a wider range of students to make effective links to their personal experience.

Teaching software design promises to provide a learning experience with three features that do not often coexist. The design process requires analytic, logical, and inductive reasoning. Designers start with an ill-structured problem. They have to make explicit the implicit requirements of that problem and then generate a pathway from goal to solution. Second, they must do so with a number of goals at the same time, which means they must integrate these disparate threads into a common structure. Third, such thinking is difficult and time consuming and students typically prefer to escape into simpler, less demanding ways of thinking, cf., [27] with structured diagramming in programming students, [3] with learning LOGO programming, [28] with algebraic story problems). In analysis and design of software students can not avoid using their most powerful reasoning skills.

The object-oriented paradigm may provide an effective basis by which to teach software design independent of a programming language because (a) its basic units--object/class structures--may be intuitively easier to grasp [29], (b) the shift from a natural-world structure in the analysis phase to an input-process-output structure common in the design phase with procedural languages may be avoidable[30] and (c) the encapsulation of the code within object/class structures permits design conceptualizations that are not based in code.

The explorations described in this paper were designed to answer the following questions. Can students with no programming experience learn to create software designs? What sorts of experience

are necessary to achieve this? Do the students' designs display systematic inadequacies? Is this sort of learning experience also useful for students who do know how to program?

2.0 Development of Instructional Protocol

We developed a short, noncredit summer course for high school students using object-oriented design techniques centering on CRC cards (Class-Responsibility-Collaboration; Beck and Cunningham [31]; Wirfs-Brock, Wilkerson, and Wiener [32]) to design computer games. The instructional protocol was developed via user-centered design process. We will discuss the reasoning behind the use of each of these components and then we will describe briefly the phases of development.

2.1 Components of Instruction As a technique for developing high-level designs, we used CRC cards. CRC cards concretize the task of parsing the problem and provide a flexible way to represent designs visually. Each card represents a class and its functions (responsibilities) are listed on the card. Either as a separate step or while writing down responsibilities, designers think through the flow of messages or collaboration. They must determine (a) what information the class needs to complete each responsibility and from what class it gets that information, and (b) where the class should send the information it generates. We used the technique of identifying nouns and verbs in the description to make a first pass at classes and responsibilities, respectively. This provides a familiar starting point for naive students.

We chose computer games as our design targets because (a) they are familiar to students, (b) they are interesting to students, and (c) their reliance on numerous concrete objects (e. g., game pieces) on the screen makes it easier to conceptualize classes than would be true for traditional computer science problems involving numbers and/or words.

Our teaching techniques are similar to the cognitive apprenticeship model described by Collins, Brown, and their collaborators [33], [34], which is based on Palincsar and Brown's method of teaching reading comprehension [35], [36], Bereiter and Scardamalia's teaching of writing [37], [38] and Schoenfeld's method for teaching mathematical problem solving [39], [40]. They include three major features common to all three of these disparate, but effective teaching techniques:

1. modeling the expert reasoning process,

2. providing scaffolding and coaching for student learning,
- and
3. having the students work in small groups.

2.2 Development Phases The development of our course has followed the general strategies of user-based design (in this case, the users were the students). At each phase of development a prototype of the instructional protocol was developed, tested, revised, and retested.

Our first development activity was to survey 22 high school students to discover what sorts of computer games they had played, liked, and would like to design. The students had had most experience with arcade games and rated them as most enjoyable, but when asked to write an essay describing the design of a computer game 78% created a realistic or a fantasy adventure game. We thus chose adventure games as our focal game type because the students' choice of what to design seemed the more important choice and because adventure games lend themselves to rapid prototyping. The students' final project would be to develop a design for their own game and we would then choose one of their games to implement. We developed a game shell in Hypercard to serve as a prototyping environment that would enable staff programmers to implement a student design.

While the game environment was being developed, we prototyped the instructional use of CRC cards. Our overall goal is to have students learn how to create software designs. Their learning of declarative knowledge is subordinated to that goal. In our first instructional prototype two groups of students were given the problem of creating a design for a computer version of the game of volleyball (as played in high school gym class). An expert gave a brief lecture and a short handout of rules and suggestions, then served as coach. Students found this task very difficult and frustrating, although they were able to produce fairly complete sets of CRC cards after much discussion and coaching. Specific problems were: (a) agreeing on the nature of the volleyball game they were designing (b) understanding the distinction between classes and responsibilities, (c) keeping straight the client-server relationship when doing collaborators, (d) abandoning the role of player (or coach) and adopting the role of the game creator who had to make everything happen, including making the ball and players move.

We also observed three groups of graduate students during their initial attempts to create CRC cards for Parnas' KWIC Index Production System [41] after having observed their instructor model a design for a restaurant management system. These students found the product they were to design difficult to understand and were able to generate reasonable classes and responsibilities only after intensive initial coaching. They never mentioned the restaurant management example.

In our second instructional prototype we simplified the task by having students create designs for simple extant computer games, we required that each design group create and agree on a description of the game before creating CRC cards, and we revised the CRC cards so that there were two columns of collaborators (one for clients and one for servers). Thus, students could choose to focus on client or server, as suited their needs; they were not required to fill in both columns. The use of an expert coach and cue cards (revised and simplified) was continued and the other components of the cognitive apprenticeship model were added. Thus, the procedure started with an expert modeling the entire process--playing an extant game (Brickles, a popular shareware game), creating a description of it, underlining nouns and using them as suggestions to create class names, underlining verbs and using them as suggestions to identify responsibilities, and identifying collaborations. Then student groups created their own designs of one of two similar games (Shuffleboard, one of the games available in Kids' Math.¹ or Artillery, a freeware game). Four groups of university students, each consisting of 2 to 5 students, completed this instructional protocol in 4-6 hours. Students were able to create creditable designs and they found the program comprehensible, interesting, and effective. Having the students design a game very similar to the one the expert designed was effective, as students spontaneously referred back to the demonstration and the coaches effectively referred back to the first design in their coaching. The revised CRC cards apparently eliminated the client-server confusion, although some groups needed to be reassured that the redundancy of the two collaborator columns was real and not a result of their confusion. Students continued to have difficulty shifting from the role of the player to the role of the designer.

For the third and current instructional prototype two changes and four additional activities were introduced. The two changes were: (a) the students played the game of Brickles themselves before the

¹KidsMath is a product of Great Wave Software, Scotts Valley, CA.

expert modeled the creation of a design for it and (b) the game of Artillery was eliminated because it played so slowly that students had difficulty grasping its structure, because it required that students be comfortable with mathematics of trajectories, and because there appeared to be less similarity to Brickles. The four added components are as follows.

First, to concretize how computers create a “world” on the screen, we developed a computer demonstration of how a game object (a ball) gets drawn at a particular point on the screen (by identifying X and Y coordinates), moves (draws itself, undraws itself, moves to new coordinates and draws itself, etc.), and knows when it hits something (by comparing X-Y coordinates). Second, a procedure to introduce inheritance (expert modeling in Brickles followed by coached student production) was developed. Third, we developed a more complex game for the students to design. The game is an adventure game in which players have to discover who the protagonist is and how to help him (the solution is to solve a jewelry theft to help an undercover cop). The process for developing this game, an adventure game, was: (a) a paper prototype was developed, (b) volunteer students “played” through a simulation of the game using the paper prototype, (c) an on-line prototype, using the game shell, was created, (d) volunteer students and staff played the game and tried to break it, (e) a final version was implemented. Fourth, students created their own design, using the same process they had been practicing with Shuffleboard and the adventure game.

3.0 Course Content

The current program consists of a tightly structured sequence that includes both (a) the staggered introduction of specific techniques (e. g., introducing inheritance as a separate and later step), (b) increasing the complexity of the problem situation, and (c) fading the learning supports as students become more proficient [33], [34], [39], [40].

During the course, students engaged in the following activities:

1. Students played an extant computer game (Brickles) and then watched an expert demonstrate how to create a design for that game with CRC cards, explicitly modeling the necessary reasoning. The expert provided brief, informal definitions of the key terms (object, class, responsibilities, collaborators), but focused on thinking-aloud descriptions of how and why he made choices rather

than on declarative exposition. He explicated alternative choices and descriptions. In the first lecture he modeled writing a description.

2. Students learned to use Microsoft Word on the Macintosh so they could write their descriptions.
3. An expert used the ball demonstration to explain how computers simulate a world. The goal was to provide sufficient concrete understanding of how computers create games that nonprogramming students could adopt the designer's point of view.
4. The expert modeled the generation of class cards, responsibilities and collaborators, finishing with an abstracted reply [39], [40].
5. Students played a second extant game (Shuffleboard) and then, in small groups created their own description and CRC cards. Coaches provided maximum scaffolding.
6. The expert modeled inheritance for Brickles.
7. The students identified inheritance relations and created cards for abstract classes in Shuffleboard.
8. Students played the adventure game and in small groups created a design for it. This activity provided not only the experience of designing a different sort of, and more complex game, but also created opportunities for discussion of reusability and extensibility. Coaches faded their support.
9. Students in small groups created designs for their own game. Coaches consulted when needed. In the summer program the class as a whole agreed on a single game to be implemented (part of the negotiation for this design was that it could be produced in the game environment).
10. Staff programmers implemented a prototype of the chosen design. Students then played the game to see that their design actually described the game. This
11. Students took a substantive test of their knowledge of object-oriented design.

4.0 Course Delivery

The current course was tested as a summer institute with high school students. Its applicability to programming students was assessed by doing the first part of the course with advanced computer science majors who were enrolled in a senior seminar on object-oriented design. This paper compares the experiences of these two groups of students².

²A fuller description of the results of the summer program can be found in [42]

The summer course was offered free to 26 high school volunteers who had been recruited through visits to the classroom of all college-preparatory mathematics classes in a local high school. Fourteen students had used a computer before, but only 4 had studied programming (BASIC); 12 students had no computer experience. The first, fourth, fifth, sixth, and seventh activities described above (Brickles demonstration and Shuffleboard design creation) were completed by 8 computer science majors enrolled in a senior seminar on object-oriented design. At the time of testing they had studied the basic concepts of object-oriented design and had begun to study Smalltalk. Four of them had been employed as part-time programmers on an object-oriented design project for 1-2 years. They were divided into two groups, one comprised of the experienced object-oriented programmers and the other comprised of students with extensive programming experience but no object-oriented experience beyond the course in which they were enrolled.

The high school students were given a written test containing 16 multiple-choice questions and 3 create-part-of-a-design questions. These questions tested understanding of the basic process of creating designs--the nature of objects, classes, responsibilities, collaborators, and inheritance--and understanding of the implications of their design activities for the software design process--notions of reusability, extensibility, and encapsulation. The multiple choice questions were submitted to an item analysis to eliminate misleading and nondiscriminating questions. Ten questions were retained and 6 additional questions were revised for the seniors' test. One of the open-ended questions was dropped for the seniors' written test because it required familiarity with the adventure game, which the seniors did not experience. A second open-ended question was revised to reduce ambiguity before it was given to the seniors.

Both sets of students were also asked a set of questions about their reactions to the program.

The summer program met for 3 weeks, 4 days a week, 3 hours per day. Students completed the entire program described above. The computer science seniors created their designs without coaching (the instructor was available, but they chose to complete the assignment independently). The seniors took the written test and the survey of their opinions after they had completed all course requirements.

5.0 Course Outcomes

5.1 Students' responses to the program. The summer program apparently held the high school students' interest. Of the 26 high school students who began the program, 23 completed it. The students' behaviors during the program also suggested interest. They typically stayed in their groups working steadily for an hour or more, even when the activity involved writing. Although the students undoubtedly volunteered because they were intrigued by the topic, they presumably would not have stayed if they had found it tremendously boring or difficult.

Students in both groups were asked how they felt about various aspects of their CRC experience. They all were asked how interesting the various components of the expert demonstration and student design work was. These results are presented in Table 1. With respect to interest, the college students found their CRC activities quite interesting. Their average rating was 4.02 out of 5.00. The high school students, in contrast, were much more neutral--their average rating was 2.63.

The comment of one senior gives a suggestion about why the activity was so interesting to the more experienced computer science majors: "Actually designing the shuffleboard game was a great way of seeing the design process as a whole activity. It also showed me that the design process was something I myself could do."

These results should not be taken to indicate that the high school students did not find the summer program profitable. They judged the overall program much more favorably than they did the initial design activities. Their average response to the question of how much they enjoyed the program was 3.57, which was nearly as favorable as the college seniors' response to the two sets of activities they experienced. Furthermore, they were not put off by the strategy of having staff programmers implement only one design. Some students wanted to design a game that could be implemented and created appropriate adventure games. Other students chose to create games that they knew could not be implemented. Once the design to be implemented was chosen, 11 students moved from their designs to the to-be-implemented game, but 9 students continued to work on their own design even though it would not be implemented.

5.2 Students' First Design. It is difficult to analyze the adequacy of sets of CRC cards because there is no one correct design. Nonetheless, the context (extant program, relatively simple design, game with objects represented on the screen) reduced the variability. Most classes coincided with

screen object--variability in choice of classes occurred primarily because (a) groups varied in how they parsed the screen objects, and (b) groups sometimes left out classes. Figure 1 depicts the screen layout and the smallest set of classes that provide a complete design, i. e., a design that would, if implemented, reproduce the game in all its essential features. We then compared the designs created by five high school groups (the BASIC group was omitted because a member inadvertently lost their design while it was in progress) and the two senior designs using the minimal design as a point of comparison. The four classes starred in the figure were included in all eight designs.

One would expect that the classes that novices would be most likely to fail to include would be those not represented by screen objects. This was true for the high school students whose omissions were mostly those that are not screen objects (see Table 2). Note also that the high school students left out classes to a greater extent than did the seniors. There was little tendency, however, to create classes that had no computer function--the only such class was player, which had no function because all interactions between human and computer game were handled by their mouse class. one high school group created a class card for player, although two other groups cited player as a collaborator.

Table 2 also displays the instances in which groups used more than one class to represent what could have been effectively represented by one class. Some of these multiple classes are just failures to see that two classes have the same responsibilities, e. g., diamond and square are both point boxes. In all other cases, students had two or more classes with one or two responsibilities in situations where a single class could easily handle all of these responses with no negative effects (and often with the positive effect of eliminating some collaborations). Often these latter types of classes appear to stem from students' tendencies to create a class for every object on the screen. For example, a puck launching class would have only the responsibilities of drawing itself and launching pucks, and the table would only have the responsibility of drawing itself and determining how sticky or slippery it was. These two classes could be more efficiently represented as one playing area with all three responsibilities. In an optimal design a responsibility should not be shared by two classes and classes with few responsibilities should be avoided [32]. Thus the multiple classes typically represent less-than-optimal design decisions. Both samples created multiple classes. In fact, the group with the most

multiple classes was a senior group--they had 18 classes all together, whereas all the other groups had 10-13 classes.

To gain an overall score of adequacy of classes, we assigned 1 point for every class that was present in the minimal design and a fraction of a point for each class that was listed with multiple classes (e.g., if .5 if 2 classes were created, .33 if three were used), and no points for player class, which described the human involvement (e.g., controls the mouse). This scheme completely described the students' classes, i. e., there were no classes that would be omitted by this scoring, and a perfect score was 10 and. The two senior groups had both the highest score (9.5) and the lowest score (6.03), and the high school groups ranged between (6.75 to 8.83).

Responsibilities have two characteristics--(a) they are present and adequately described and (b) they are assigned to reasonable classes. Although evaluation of the presence of a responsibility is generally straightforward, some responsibilities (e. g., control functions in Shuffleboard) may function well in several different classes. Thus, we evaluated only whether each group included an adequate representation of each responsibility.

We identified 37 correct responsibilities and calculated the percent included in each design. The average percent included in the seniors' designs was 80 percent and in the high school students' designs 62 percent. In fact the distributions were nonoverlapping--both senior groups did better than any high school groups. The seniors were particularly superior at the less obvious--starting and ending the game, calculating the score, etc.

The adequacy of the descriptions of responsibilities was evaluated by whether they truly described a function rather than a class or a collaborator. Thus, this analysis evaluates the extent to which groups confused classes, responsibilities, and collaborators. The seniors made no such errors, whereas for 3 high school groups their responsibilities sounded like descriptions of the classes they were in (on 5 to 8 occasions each) and for 2 other groups their responsibilities sounded like collaborators (on 1 and 3 responsibilities, respectively).

The seniors created many more responsibilities than did the high school students. The two senior groups included an average of 20 extra responsibilities (not included in the 37 we had identified as correct), whereas the high school students' comparable average was 3.4. Of the seniors' extra

responsibilities, half described functions that should be postponed until later stages of the design (either initialization or data flow functions) and half were redundant with a responsibility we had marked as correct. Only 1 high school group had more than 2 extra responsibilities, and 6 of their 10 extra responsibilities were repetitions of the responsibility “Knows location” in inappropriate classes, which we think represents an inappropriate generalization from the Brickles demonstration, in which most screen objects had to communicate their location with another class.

To analyze collaborators, we first looked for violations of rules of collaboration. First, all classes must have collaborators; otherwise, the functions of that class serve no purpose in the overall program. Four groups violated this rule on one class each--3 summer and 1 senior group. The second type of error was collaborating with a nonexistent class. Four of the five high school groups and one of the two senior groups committed this error, and the fifth high school group had collaborations with their player class, which had no computer functions.

To assess whether students were able to think through the path of collaborations, we examined collaborations among the classes responsible for scoring. The games' scoring is as follows. At the end of every round (nine pucks) the player's actual score is calculated (based on the number of pucks on the 10-point and 1-point boxes) and the player enters his/her own calculations into the number grid (see Figure 1). The scores are compared and the player given feedback about the correctness of his/her calculation in the message box. We assessed whether the collaborations specified were sufficient for each class to fulfill its responsibilities (different design groups had assigned the responsibilities for calculation of score and the comparison of game's score to player's score to different classes). Both senior groups had complete sets of collaborators, whereas three of the five high school groups exhibited some confusion.

We also analyzed how the student design groups managed collaborations necessary for starting the game. Again the two senior groups both provided complete collaborations. Only two high school groups included a game class and only one of these groups specified a complete set of relevant collaborations. The three groups without a game class were all incomplete and/or confused. Thus, it appears clear that the college seniors were adept at considering data flow issues, but that some nonprogramming groups could manage this with no programming background.

Analysis of students' inheritance productions are not included because the productions for the two samples were so different. The high school students produced CRC cards for abstract classes and for the most part did not create class hierarchies, whereas the senior groups produced tree diagrams with no CRC cards for abstract classes. The absence of CRC cards in the seniors' designs made it difficult to assess the quality of the inheritance diagrams. Our impression is that one group created what was essentially a data-flow diagram and the other a Booch diagram.

5.3 Written test. To compare the high school and college students we first inspected the 6 questions that had been revised after the high school students took the test. Four of those questions were deleted, because the revision had made them weaker psychometrically. Thus, the analyses were based on 12 multiple choice questions and the two partial design questions that were given to both samples.

One partial design question asked the students to fill in an inheritance diagram. Seniors scored 68% correct compared to the high school students' 48%. The second question gave the main classes for a game of tic-tac-toe and asked students to fill in the responsibilities and collaborators on CRC cards. The seniors scored much higher (76% correct compared to 57% for the high school students), although part of this difference may be attributable to changes in wording designed to make the question clearer.

The results of the written test were submitted to a 2 factor analysis of variance with groups (high school vs. college seniors) as a between-subjects factor and test component (multiple choice vs. partial design) a within subjects factor. Because the variances were decidedly heterogeneous and the sample sizes were substantially different, the alpha level of .01 was used. On the 12 multiple choice questions repeated from the summer test, the seniors scored on average 90% correct compared to the high school students' 56%. This difference was significant, $F(1,29) = 10.51$, $p < .01$. The students did no better on the multiple choice questions than they did on the partial design questions, $F(1,29) = 4.81$, NS. The interaction was not significant, $F(1, 29) = 2.28$. Thus, the college seniors were superior in both declarative knowledge, as assessed by the multiple choice questions, and in ability to apply what they knew, as assessed by the partial-design questions.

Although the seniors did substantially better than the high school students on average, some high school students did very well. The highest scoring high school student did better than 5 of the college seniors and 9 high school students did better than the worst college senior. Thus, the program seems appropriate for both levels of expertise.

5.4 The Role of Programming Experience. Clearly the college seniors did better than the high school students on most dimensions. Does this superiority result from their programming experience, their computer science knowledge, their advanced age and its related intellectual and emotional maturity? Two aspects of these data can provide some information about the specific role of programming experience. Four of the high school students had taken BASIC programming instruction. Their average performance on the test (multiple choice questions plus a weighted average of the partial design questions) was 11.69 compared to 9.57 for the nonBASIC students. We calculated a t test on these results, even though the unequal n's make the analysis vulnerable to an increase in Type I error. The difference was nonsignificant, $t(21) = 1.15$, so we have no reason to believe there was any difference between the BASIC students and the other high school students.

We also compared the seniors who had extensive object-oriented programming experience to those whose only experience in programming object-oriented languages was in this course (Smalltalk). The more experienced seniors' score was 16.27 compared to 13.85 for the less experienced seniors. We calculated a t test, although the sample size was too small for it to be conclusive. The difference was not significant, $t(6) = 2.18$.

5.5 Student feedback about effectiveness of instruction The design activities appeared to be well within the grasp of both high school students and college seniors. The students were asked how understandable each component of the instructional protocol was. Their ratings (on a 1 to 5 scale, with 1 being the lowest rating) are given in Table 1. The average rating of comprehensibility was 3.72 for high school students and 3.68 for college seniors. No high school student or college senior reported any activity other than inheritance as "very confusing," and only 1 or 2 high school students found inheritance very difficult. Two seniors commented that they needed more work with inheritance. Three seniors made comments about collaborators. They agreed that the CRC cards did not help (and maybe

interfered) with finding collaborators. Two suggested that a visual representation that facilitates the forming of a mental image of the classes interacting with each other would be useful.

The college seniors were asked to comment on the small group activity. They made the following observations: it is more interesting, it is easier, groups produce a better product, more thinking, and a more thorough design. They also reported that the small groups were frustrating, because of the need to compromise and the difficulty of getting group members together.

6.0 Discussion

This study is obviously exploratory rather than experimental. By the criteria of strict experimental design it has three major shortcomings. First, the two groups were not given the same experimental procedures and there was no control group against which to compare their performance. Second, the scoring of the designs and partial designs are open to interpretation. Third, the statistical analyses are inconclusive because of the sampling problems.

Nonetheless, the results were quite promising and the multiple samples studied provide partial replication with quite different populations. In general, both high school students and senior computer science majors learned from their design experiences--they produced creditable first designs and were able to answer test questions and to apply what they had learned to create a part of a design for a new game. In addition, both groups found these experiences valuable and not overly frustrating.

6.1 The Senior Computer Science Majors The college seniors clearly had an impressive grasp of the declarative knowledge of object-oriented design, as evidenced by their performance on the multiple choice test, but that knowledge was not sufficient to produce optimal designs, i. e., designs that would completely specify essential aspects of a game. For example, on the Shuffleboard design they had only 74% of the necessary responsibilities and on the generalization partial design, they had 76% of the responsibilities and collaborators needed.

In addition to overall better performance, the computer science majors behaved differently from the high school students in two ways. First, they were much better at uncovering nonobvious functions, particularly classes that had no screen object and control processes. Second, they tended to include an inappropriate level of detail in what were supposed to be high-level designs. Their extra

responsibilities--both the redundant ones and the low-level details--are inconsistent with principles of good design. They also were inefficient--although we did not specifically time their work periods, casual observation was sufficient to conclude that one college group spent many hours more than all other groups. This suggests that students with programming background need guidance as to appropriate level of detail in high-level design. Such guidance would fit nicely into an extension of our instructional protocol.

Finally, the students themselves found it very worthwhile. Presumably if they had been learning design in the traditional computer science curriculum, this should have been a trivial exercise--after all, it had been developed as an educational experience for completely naive high school students. This suggests that what students learn in the standard computer science curriculum does not generalize automatically to issues of object-oriented design. This is particularly striking because the students in this course had been addressing issues of design within the object-oriented paradigm. Nonetheless when it came time to apply what they had learned to creating even a very simple design, they still found it challenging.

6.2 The High School Students Even when one compares their performance to senior computer science majors, the high school students' performance is quite impressive. These students were 5 to 8 years younger than the university students, most of them had had no computer science courses compared to the seniors who were in the last year of their major, and they were participating in a voluntary summer experience rather than a credit-earning course. Yet their Shuffleboard designs were of the same general quality as those of the college seniors and their test performance was substantially above chance.

6.3 The Role of Programming Experiences This research was not designed to test the effects of knowing a programming language on learning design in a nonprogramming context. The two comparisons we could make between students with differential programming experience failed to find significant differences, but because of the small sample size we can have little confidence in these results.

It is tempting to attribute the differences in design between the high school students and computer science majors to the programming experience of the latter group, but other differences

between the groups might well account for these effects. One would hope that college seniors generally can think things through more carefully than high school students can.

7.0 Implications and Future Directions

This project demonstrates the feasibility and desirability of teaching high-level design skills both to novice students and to computer science majors. The high-school students could grasp the basic concepts necessary to create high-level designs even though they had no programming experience. The computer science majors, albeit better than the high school students at understanding the design process and thinking through the design problem, had difficulty keeping at an appropriate level of design. This difficulty has been documented in other research contexts [9], [10], [42], but it is particularly striking in this setting because these computer science majors all had extensive programming experience within their curriculum and had spent much attention during this senior seminar studying issues of design. Thus, the findings of this study suggest that learning the nature of different levels of design needs to be addressed explicitly and experientially.

Although we did not systematically test the role of object-oriented design against other design methodologies, we believe that the combination of object-oriented methodology and use of extant computer games was key to its success. CRC cards permitted students to conceptualize design in familiar terms--objects that had jobs to do and other objects to collaborate with--until they developed some understanding of how programs work. The extant computer games gave them the objects they could identify as classes.

This project also showed that the students are not necessarily driven by a need to implement their designs. Freeing instruction from the necessity of implementing every project provides more time for doing and reflecting on issues of design and design quality. One positive outcome could be providing students with more experience with a greater variety of designs and applications, which may be a crucial ingredient to success in professional designing, c. f., for example, [9], [15].

If continuing work confirms the results of this project, computer scientists will be able to reconsider the structure of the computer science curriculum. Students who learn to design before they

learn to program may well learn to program more rapidly and more effectively because they have a context in which to understand coding issues.

This project is only a beginning. We are currently working through additional issues of instruction. Among the issues we are exploring are (a) can students apply what they have learned by designing computer games to other types of applications and what contributes and/or inhibits this transfer, (b) what are effective representations to check CRC cards and to continue the design process (e. g., inheritance hierarchy, Booch diagram, collaboration graph) (c) how can programming students learn to distinguish between high and low levels of design, and (d) how can novice designers develop the ability to see where a design is heading and why some directions are better than others i. e., become reflective practitioners.?

References

- [1] J. Dalbey & M. C. Linn, "The demands and requirements of computer programming: A review of the literature" Journal of Educational Computing Research, Vol.1, 1985, pp. 253-274.
- [2] H. Gorman, Jr. & L. E. Bourne, Jr., "Learning to think by learning Logo: Rule learning in third grade computer programmers" Bulletin of the Psychonomic Society, Vol. 21, 1983, pp.165-167.
- [3] D. M. Kurland, R. D. Pea, C., Clement, & R. Mawby, "A study of the development of programming ability and thinking skills in high school students" Journal of Educational Computing Research, Vol. 2, 1986, pp. 429-458.
- [4] M. C. Linn & J. Dalbey, "Cognitive consequences of programming instruction: Instruction, access, and ability" Educational Psychologist, Vol. 20, 1985, pp. 191-206.
- [5] E. B. Mandinach & M. C. Linn, "Cognitive consequences of programming: Achievements of experienced and talented programmers" Journal of Educational Computing Research, Vol. 3, 1987, pp. 53-72.
- [6] National Assessment of Education Progress, Computer competence: The first national assessment. Princeton, NJ: Educational Testing Service, 1988.
- [7] R. D. Pea & D. M. Kurland, "On the cognitive effects of learning programming" New Ideas in Psychology, Vol. 2, 1984, pp. 137-168.
- [8] R. T. Putnam, D. Sleeman, J. A. Baxter, & L. K. Kuspa, "A summary of misconceptions of high-school BASIC programmers", Studying the novice programmer. Edited by E. Soloway & J. C. Spohrer, Erlbaum, Hillsdale, NJ, 1989, pp. 301-314.
- [9] R. S. Rist, "Plans in programming: Definition, demonstration, and development", Empirical studies of programmers. Edited by E. Soloway & S. Iyengar, Ablex, Norwood, NJ, 1986, pp. 28-47.
- [10] R. S. Rist, "Schema creation in programming" Cognitive Science, Vol. 13, 1989, pp. 389-414.
- [11] E. Soloway, "Learning to program = learning to construct mechanisms and explanations" Communications of the Association for Computing Machinery. Vol. 29, 1986, pp. 850-858.

- [12] E. Soloway, & K. Ehrlich, "Empirical studies of programming knowledge" IEEE Transactions on Software Engineering, Vol. SE-10, 1984, pp. 595-609.
- [13] B. Adelson, & E. Soloway, "The role of domain experience in software design" IEEE Transactions on Software Engineering, Vol. 11, 1985, pp. 1351-1360.
- [14] M. E. Atwood & H. R. Ramsey, Cognitive structures in the comprehension and memory of computer programs: An investigation of computer debugging (Tech. Rep. TR-78-A210). U. S. Army Research Institute for the Behavioral and Social Sciences, Alexandria, VA, 1978.
- [15] R. Guindon, "Designing the design process: Exploiting opportunistic thoughts" Human-Computer Interaction, Vol. 5, 1990, pp. 305-344.
- [16] R. Jeffries, A. A. Turner, P. G. Polson, & M. E. Atwood, "The processes involved in designing software", Cognitive skills and their acquisition. Edited by J. R. Anderson, Erlbaum, Hillsdale, NJ, 1981, pp. 255-283.
- [17] E. Kant, & A. Newell, "Problem solving techniques for the design of algorithms" Information Processing and Management, Vol. 28, 1984, pp. 97-118.
- [18] M. C. Linn, "Fostering equitable consequences from computer learning environments" Sex Roles, Vol. 13, 1985, pp. 229-240.
- [19] W. Swartout, & R. Balzer, "On the inevitable intertwining of specification and implementation" Communications of the Association for Computing Machinery, Vol. 25, 1982, pp. 438-440.
- [20] J. L. Connell & L. B. Shafer, Structured rapid prototyping: An evolutionary approach to software development. Prentice-Hall, Englewood Cliffs, NJ, 1989.
- [21] O. J. Dahl, E. W. Dijkstra, & C. A. R. Hoare, Structured programming. Academic, New York, 1972.
- [22] S. Hekmatpour & D. Ince, Software prototyping, formal methods and VDM. Addison-Wesley, Reading, MA, 1988.
- [23] H. D. Mills, "Structured programming: Retrospect and prospect" IEEE Software, Vol. 3, November 1986, pp. 58-66.
- [24] D. L. Parnas & P. C. Clements, "A rational design process: How and why to fake it" IEEE Transactions on Software Engineering, Vol. 12, 1986, pp. 251-257.

- [25] N. Wirth, "Program development by stepwise refinement" Communications of the Association for Computing Machinery, Vol. 14, 1971, pp. 221-227.
- [26] P. J. Denning, D. E. Comer, D. Gries, M. C. Mulder, A. Tucker, A. J. Turner, & P. R. Young, "Computing as a discipline" Communications of the Association for Computing Machinery, Vol. 32, 1989, pp. 9-22.
- [27] J. Dalbey, F. Tourniaire, & M. C. Linn, "Making programming instruction cognitively demanding: An intervention study" Journal of Research in Science Teaching, Vol. 23, 1986, pp. 427-436.
- [28] J. E. Sims-Knight, Teaching Students to Use Mathematics: Eliminating Errors in Mapping from Natural Representational Systems to the Abstract Symbol Systems of Mathematics. Final Report of NSF Grant MDR 84-10316, 1990.
- [29] M. B. Rosson & S. R. Alpert, "The cognitive consequences of object-oriented design", Human-Computer Interaction, 1990, Vol. 5, pp. 345-379.
- [30] M. B. Rosson & E. Gold, "Problem-solution mapping in object-oriented design", October 1989, OOPSLA '89 Proceedings, pp. 7-10.
- [31] K. Beck & W. A. Cunningham, "A laboratory for teaching object-oriented thinking", OOPSLA '89 Proceedings, 1989, 1-6.
- [32] R. Wirfs-Brock, B. Wilkerson, & L. Wiener, Designing object-oriented software. Prentice-Hall, Englewood Cliffs, NJ, 1990.
- [33] A. Collins, J. S. Brown, & A. Holum, "Cognitive apprenticeship: Making thinking visible." American Educator, Winter, 1991, 6-11, 38-46.
- [34] A. Collins, J. S. Brown, & S. E. Newman, "Cognitive apprenticeship: Teaching the crafts of reading, writing, and mathematics", Knowing, learning, and instruction: Essays in honor of Robert Glaser. Edited by L. B. Resnick, Erlbaum, Hillsdale, NJ, 1989, pp. 453-494.
- [35] A. S. Palincsar & A. L. Brown, "Reciprocal teaching of comprehension-fostering and monitoring activities" Cognition and Instruction, Vol. 1, 1984, pp. 117-175.

- [36] A. L. Brown & A. S. Palincsar, "Guided cooperative learning and individual knowledge acquisition", Knowing, learning, and instruction: Essays in honor of Robert Glaser. Edited by L. B. Resnick, Erlbaum, Hillsdale, NJ, 1989, pp. 393-451.
- [37] C. Bereiter & M. Scardamalia, The psychology of written composition. Erlbaum, Hillsdale, NJ, 1987.
- [38] M. Scardamalia, C. Bereiter, & R. Steinbach, "Teachability of reflexive processes in written composition" Cognitive Science, Vol. 8, 1984, pp. 173-190.
- [39] A. H. Schoenfeld, Problem solving in the mathematics curriculum: A report, recommendations, and an annotated bibliography. (M.A.A. Notes #1). Mathematical Association of America, Washington, DC, 1983.
- [40] A. H. Schoenfeld, Mathematical problem solving. Academic Press, Orlando, FL, 1985.
- [41] D. L. Parnas, "On the criteria to be used in decomposing systems into modules" Communications of the Association for Computing Machinery, 1972, December, pp.1053-1058.
- [41] J. E. Sims-Knight & R. L. Upchurch, Teaching software design to high school students. Unpublished manuscript, 1993.
- [42] M. C. Linn & M. J. Clancey, "Can experts' explanations help students develop program design skills?" International Journal of Man-Machine Studies, Vol. 36, 1992, 511-551.

Table 1

Average Ratings of Two Groups on Expert Demonstration and Students' First Design Activity

	How Interesting		How Understandable	
	High School	CS Seniors	High School	CS Seniors
Demonstration				
Description	3.00	4.10	4.00	4.60
CRC Cards	2.30	3.90	3.30	3.80
Inheritance	2.80	4.00	3.70	3.20
Students' Design				
Description	2.60	4.00	3.90	3.90
CRC Cards	2.40	4.10	3.70	3.50
Inheritance	2.70	4.00	3.70	3.10

Table 2

Omitted and Multiple Classes in Students' Design for Shuffleboard

No. Groups who failed to include class		
Class	No. High School Groups. (of 5)	No. Senior Groups (of 2)
Number grid	1	1
Mouse	3	0
Game	3	0
No. Groups who used multiple classes		
Class	No. High School Groups (of 5)	No. Senior Groups (of 2)
Message Box Class	1 (2 classes)	1 (3 classes)
Info Box	0	1 (2 classes)
Sticky/slippy Meter	1 (2 classes)	1 (2 classes)
Playing Area	5 (2, 3, 3, 3, 4 classes)	2 (2, 5 classes)
Mouse	0	1 (2 classes)
Game	1 (2 classes)	0