

Sims-Knight, J. E., & Upchurch, R. L. (1992). Teaching object-oriented design to nonprogrammers: A progress report. Proceedings of OOPSLA-92 Educators' Symposium. Vancouver, British Columbia, Canada.

Teaching Object-Oriented Design to Nonprogrammers

Judith Sims-Knight, Ph.D.**
Department of Psychology
University of Massachusetts at Dartmouth
Old Westport Road
Dartmouth, MA 02747
(508-999-8896)
JSIMSKNIGHT@UMASSD.EDU

&

Richard Upchurch, Professor
Department of Computer and Information Science
University of Massachusetts at Dartmouth
Old Westport Road
Dartmouth, MA 02747
(508-999-8295)
RUPCHURCH@UMASSD.EDU

** Primary Contact Person

Teaching Object-Oriented Design to Nonprogrammers¹

Judith E. Sims-Knight

Richard L. Upchurch

University of Massachusetts Dartmouth

New approaches to teaching introductory computer science are sorely needed. Current instruction in programming has been shown to result in 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., Dalbey & Linn, 1985; Gorman & Bourne, 1983; Kurland, Pea, Clement, & Mawby, 1986; Linn & Dalbey, 1985; Mandinach & Linn, 1987; National Assessment of Educational Progress, 1988; Pea & Kurland, 1984; Putnam, Sleeman, Baxter, & Kuspa, 1989; Rist, 1986, 1989).

An alternative approach, if feasible, would be to teach students software design before teaching them a programming language. This approach makes sense in a number of ways:

- Design is one of the three central processes in the core of computer science (Denning et al., 1989).
- Teaching software design would result in a broader introduction to computer science.
- Progress in learning analysis and design should be more effective and more rapid if students can focus on those issues rather than having first to master the complex procedural skills of a programming language.
- If one is freed from the necessity of selecting programming tasks simple enough for novice coders, one can select activities and examples that are more interesting, thus enabling a wider range of students to make effective links to their personal experience. Such activities can also be more complex.

The proposed approach is made feasible by the development of the object-oriented paradigm, because (a) its basic units--object/class structures--can be intuitively grasped, (b) once the object/class structures are identified, they do not need to be transformed into code-based conceptualization (as is characteristic of the analysis-design shift in traditional programming), and (c) the encapsulation of the code within object/class structures permits design conceptualization that are not based in code.

Teaching object-oriented design promises to provide a learning experience with three features that do not often coexist. The design process requires analytic, logical, and integrative 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. In addition, 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. Such thinking is difficult and time consuming and students typically prefer to escape into simpler, less demanding modes of functioning (c. f., Dalbey, Tourniaire, & Linn, 1986, with structured diagramming in programming students, Kurland et al., 1986, with learning LOGO programming, Sims-Knight, 1990, with algebraic story problems). In analysis and design of software this process takes center stage and can not be circumvented.

¹ This research is supported by National Science Foundation Grant No. MDR-9154008.

We are engaged in a "proof-of-concept" study to answer the following questions:

- Can naive high school students learn to create object-oriented designs? What sorts of experience are necessary to achieve this?
- Do the students' designs differ from those of individuals who know how to program? If so, in what ways? Can any systematic inadequacies be corrected?
- To what extent are the students developing generalizable design skills? This will be measured by their ability to design (a) more complex software of the same sort as the software they learned on (i. e., more complex computer games), and (b) software that differs substantially in domain (e.g., software tools that process words or pictures).

Initial Teaching Strategies

Guidance for teaching design is sparse. We were, however, determined to use the existing knowledge where ever possible.

Research on the cognitive processes of expert designers (Adelson & Soloway, 1985; Atwood & Ramsey, 1978; Guindon, 1990; Jeffries, Turner, Polson, & Atwood, 1981; Kant & Newell, 1984; Linn, 1985; Rist, 1986; Swartout & Balzer, 1982) and insights from software engineers (Connell & Shafer, 1989; Dahl, Dijkstra, & Hoare, 1972; Hekmatpour & Ince, 1988; Mills, 1986; Parnas & Clements, 1986; Wirth, 1971) suggest that:

- 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.
- 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.

We decided to adapt the technique of CRC cards (Class-Responsibility-Collaboration) as described by Beck and Cunningham (1989) and by Wirfs-Brock, Wilkerson, and Wiener (1990). We chose CRC cards because (a) they concretize the task of parsing the design, thereby providing a technique that permits beginners to get started, and (b) they provide a flexible way to represent designs visually in a single representational system.

We have chosen computer games as our design targets because (a) they are familiar to students, (b) they are interesting to students, and (c) they are easier to conceptualize as object classes than are traditional computer science problems involving numbers and/or words. We start out with extant computer games that the students play and then create a design representation for. After their skills are developed in this context, they will design their own games.

The students work in small groups for three reasons. First, a large part of the intellectual work in the early phases of software design involves uncovering implicit goals. Thus, groups can take advantage of Malhotra, Thomas, Carroll, and Miller's (1980) finding that discussing proposed solutions helps clarify both implicit goals and the parameters of solutions. Second, the verbal interactions between team members externalize at least some of their otherwise hidden thinking. Third, group work permits students to differentiate the roles of (a) generator of ideas, (b) critic of ideas, and (c) moderator of progress, which need to be internalized for students to be effective problem solvers (Collins, Brown, & Holum, 1991), although the research exploring whether groups or individuals learn better has been mixed (e. g., Rohwer

& Thomas, 1989; Slavin, 1987; Trowbridge, 1987; Webb, 1984), it is clear that groups typically exhibit better learning when the outcome depends on the performance of each individual within the group. With three-member teams, we can ensure that this condition is met.

Research Strategy

During the academic year 1991-1992 we are conducting a series of preliminary investigations to develop our teaching techniques as well as measures of effectiveness. During the summer of 1992 we will run a summer institute for high school students.

Preliminary Investigations

Method. We have currently run 6 groups of university students, each consisting of 2 to 5 students. Two of the groups developed designs for two games, the other four groups first observed an expert designing one game and then developed another in their group. One of the groups consisted of computer science majors, the other groups were all nontechnical majors without programming experience. The last four groups were given questionnaires to complete individually at home. Of the 15 students, 11 returned the questionnaire.

In addition we observed three groups (of 3 students each) in a graduate course in design develop their first CRC-card-based design.

We used a variety of teaching techniques as we explored ways to facilitate learning. Our basic strategy was to identify where students had difficulty and to develop strategies to facilitate learning.

Results concerning teaching. These preliminary investigations produced the following insights about the teaching of object-oriented design:

- Demonstrating the use of CRC cards using a vastly different domain (demonstration of restaurant management to bibliographic index) renders the demonstration useless. The students do not make the connection from the demonstrated example to their own work. Using closely related examples (demonstration of one simple arcade-type computer game--Brickles--to another--Shuffleboard or Artillery-- results in two types of transfer. First, students spontaneously refer back to the demonstration when developing their own design. Second, when a coach suggests they think about how something was done in the demo, students can sometimes recall the appropriate aspect of the demonstration and use that information to understand the analogous aspect of their own design.
- When students had to create a CRC-card-based design with only a brief exposition on how to do it (i. e., without a demonstration and without coaching), they had significant problems that did not seem to occur with students who had seen a demonstration. They found it difficult to differentiate between classes and responsibilities and between responsibilities and collaborations. In other words, their understanding of the basic three components of the design was muddled and confused. Their descriptions of responsibilities often did not focus on functions and were so vague that they had continuing difficulty understanding the meaning of their own words. The five students unanimously recommended that an initial demonstration be provided so that they would have an overview of the process before they began. They also all felt that a coach was a desirable feature.
- Students in our first two groups got confused between client and server roles when identifying collaborators. We therefore revised the CRC cards so that there were two columns of collaborators and both client and server classes could be identified simultaneously (even though this is redundant in that each collaboration is

identified on two cards) (see Table 1). Students in the last four groups used the revised cards successfully. There was little confusion between the client and server roles and the students did not comment negatively on having to do the same thing twice.

- When students were provided cue cards to serve as reminders (see Table 2), they rarely referred to them and even more rarely used the information therein. When an expert coach brought up the rules at appropriate times, the students were able to use the cues. Thus, students can use such rules, but need to be provided the scaffolding of a coach to do so.
- Not one of our groups ever spontaneously ran through a scenario to check their designs. When an expert coach instructed them to create a scenario and use it to check the cards, the students were able to do so. Again scaffolding needs to be provided.
- The only substantial difference between students with and without programming experience was on consideration of aspects of the design that are specific to the computer game, e. g., the nature of movement, hitting, "sticky" surfaces. Students without programming experience (a) did not spontaneously think about such issues, (b) did not respond to cues concerning them, and (c) learned to assign such responsibilities (e.g., knows its positions, knows when it has been hit) only with substantial coaching. Our one computer science student group spontaneously thought of these issues and could reason out the assignment of responsibilities fairly independently. In subsequent pilot testing, we increased emphasis on this issue. Students then were able to create the appropriate responsibilities with the help of the coach, but we believe that they would be unable to do so on their own. In the next trial to be run later this Spring, we will produce a demonstration of how objects in software know their position (by X-Y coordinates), move (animation techniques), and know when they hit something (by comparing X-Y coordinates). Hopefully, this will provide the concrete understanding of how computers operate that we think essential to designing software still without requiring that students learn code. If accepted, the results of this exploration will be included.

The resulting learning procedure conforms in many respects to the cognitive apprenticeship approach that Collins, Brown, and Newman (1989) advocate on the basis of the work of Palincsar and Brown's method of teaching reading comprehension (1984, Brown & Palincsar, 1989), Bereiter and Scardamalia's teaching of writing (1987; Scardamalia, Bereiter, & Steinbach, 1984), and Schoenfeld's method for teaching mathematical problem solving (1983, 1985). It includes two of the three major features common to all three of these disparate, but effective teaching techniques--modeling the expert reasoning process, and providing scaffolding and coaching for student learning. The procedure was not extensive enough to implement the third major feature--fading the learning supports as students become more proficient (e. g., the cue cards, the scenarios, the descriptions)--although we plan to do that later. Furthermore, we, like Schoenfeld, will eventually provide a tightly structured sequence that involves both (a) the staggered introduction of specific techniques (e. g., introducing inheritance later) and (b) increasing the complexity of the problem situation.

Results concerning learning. Students were able to produce creditable designs. Those who experienced the demonstration and coaching were able to do so on their first attempt during severe time constraints. Table 1 shows a design created by one group after only 50 minutes of work. They did not have time to check their designs for completeness nor to run through any scenarios and yet they captured the essence of most of the games they modeled. Their errors were mostly of omission and

there is no reason to suspect that they would not have been incorporated had there been more time.

We presented the concept of inheritance to only two groups, who, after some initial confusion, were able to apply the concept to a new design. Later this Spring we will run another pilot investigation that will include inheritance to verify this conclusion.

The 11 students who returned their questionnaires found the process mildly to very interesting and comprehensible. They were asked to rate on a 5-point scale the demonstration and student design activity as to its comprehensibility, interest, and effectiveness. The average rating was 3.76 (where 5 was the highest rating) and the modal response was 4. Of the 66 responses (6 questions for each student) none were the lowest rating and only 6 were the second lowest rating.

These 11 students were also asked to create a design for tic-tac-toe (individually). Only 1 of these designs was off the right track. and 3 students created designs that were complete and adequate in their essentials in only 30-50 minutes (see Table 3 for one of these). The other 7 students identified creditable sets of classes and identified a substantial subset of the responsibilities, but omitted or were vague about a major function.

Summer Program

The summer program will include the already-developed procedures and will go beyond. Its components are:

- Expert demonstrates how to design computer game, explicitly modeling reasoning and demonstrates how computers work, as discussed above.
- Students in small groups design a second, similar design. Coaches provide maximum scaffolding.
- Students in small groups play and create a design for a complex computer game. Coach fades scaffolding.
- Students in small groups create designs for their own game. The class as a whole will agree on a single game to be implemented (part of the negotiation for this design must be that it will fit into the game environment the staff used to create the complex game.
- Staff programmers will implement this design. This will then be used as a lesson in reusability and extendibility.
- Students will individually create a design of a software tool, as well as take a substantive test of their knowledge of object-oriented design.

Conclusions

This project is demonstrating the feasibility of using the object-oriented paradigm to teach nonprogramming students to develop designs. Our preliminary investigations have demonstrated that students can indeed learn to use CRC cards to produce creditable preliminary designs in a relatively short time. In addition, they generally find the process interesting and relatively painless.

Our explorations have also yielded a number of principles to maximize the learning of object-oriented design.

- A demonstration of an expert modeling his/her thought processes while producing a design similar to the students' first attempts reduces confusion and frustration in the students' first design.

- Small groups (three seems best) with a coach providing the appropriate scaffolding is an environment in which students can develop their problem-solving skills in a generally nonfrustrating and productive manner.
- CRC cards, especially when modified to clarify the client-server relation, are an effective tool.
- Cue cards and scenarios are effective aids, but the coach needs to remind students to use them at appropriate times.
- Nonprogramming students need to learn some of the basic strategies by which computers solve problems, but they can probably develop this knowledge at a more general level than the code level.

References

- Adelson, B., & Soloway, E. (1985). The role of domain experience in software design. IEEE Transactions on Software Engineering, 11, 1351-1360.
- Atwood, M. E., & Ramsey, H. R. (1978). Cognitive structures in the comprehension and memory of computer programs: An investigation of computer debugging (Tech. Rep. TR-78-A210). Alexandria, VA: U. S. Army Research Institute for the Behavioral and Social Sciences.
- Beck, K., & Cunningham, W. (1989). A laboratory for teaching object-oriented thinking. OOPSLA '89 Proceedings, 1-6.
- Bereiter, C., & Scardamalia, M. (1987). The psychology of written composition. Hillsdale, NJ: Erlbaum.
- Brown, A. L., & Palincsar, A. S. (1989). Guided cooperative learning and individual knowledge acquisition. In L. B. Resnick (Ed.), Knowing, learning, and instruction: Essays in honor of Robert Glaser (pp. 393-451). Hillsdale, NJ: Erlbaum.
- Collins, A., Brown, J. S., & Holum, A. (1991, Winter). Cognitive apprenticeship: Making thinking visible. American Educator, 6-11, 38-46.
- Collins, A., Brown, J. S., & Newman, S. E. (1989). Cognitive apprenticeship: Teaching the crafts of reading, writing, and mathematics. In L. B. Resnick (Ed.), Knowing, learning, and instruction: Essays in honor of Robert Glaser (pp. 453-494). Hillsdale, NJ: Erlbaum.
- Connell, J. L., & Shafer, L. B. (1989). Structured rapid prototyping: An evolutionary approach to software development. Englewood Cliffs, NJ: Prentice-Hall.
- Dahl, O. J., Dijkstra, E. W., & Hoare, C. A. R. (1972). Structured programming. New York: Academic.
- Dalbey, J., & Linn, M. C. (1985). The demands and requirements of computer programming: A review of the literature. Journal of Educational Computing Research, 1, 253-274.
- Dalbey, J., Tourniaire, F., & Linn, M. C. (1986). Making programming instruction cognitively demanding: An intervention study. Journal of Research in Science Teaching, 23, 427-436.
- Denning, P. J., Comer, D. E., Gries, D., Mulder, M. C., Tucker, A., Turner, A. J., & Young, P. R. (1989). Computing as a discipline. Communications of the Association for Computing Machinery, 32, 9-22.
- Gorman, H., Jr., & Bourne, L. E., Jr. (1983). Learning to think by learning Logo: Rule learning in third grade computer programmers. Bulletin of the Psychonomic Society, 21, 165-167.

- Guindon, R. (1990). Designing the design process: Exploiting opportunistic thoughts. Human-Computer Interaction, 5, 305-344.
- Hekmatpour, S., & Ince, D. (1988). Software prototyping, formal methods and VDM. Reading, MA: Addison-Wesley.
- Jeffries, R., Turner, A. A., Polson, P. G., & Atwood, M. E. (1981). The processes involved in designing software. In J. R. Anderson (Ed.), Cognitive skills and their acquisition (pp. 255-283). Hillsdale, NJ: Erlbaum.
- Kant, E., & Newell, A. (1984). Problem solving techniques for the design of algorithms. Information Processing and Management, 28, 97-118.
- Kurland, D. M., Pea, R. D., Clement, C., & Mawby, R. (1986). A study of the development of programming ability and thinking skills in high school students. Journal of Educational Computing Research, 2, 429-458.
- Linn, M. C. (1985). Fostering equitable consequences from computer learning environments. Sex Roles, 13, 229-240.
- Linn, M. C., & Dalbey, J. (1985). Cognitive consequences of programming instruction: Instruction, access, and ability. Educational Psychologist, 20, 191-206.
- Malhotra, A., Thomas, J. C., Carroll, J. M., & Miller, L. A. (1980). Cognitive processes in design. International Journal of Man-Machine Studies, 12, 119-140.
- Mandinach, E. B., & Linn, M. C. (1987). Cognitive consequences of programming: Achievements of experienced and talented programmers. Journal of Educational Computing Research, 3, 53-72.
- Mills, H. D. (1986, November). Structured programming: Retrospect and prospect. IEEE Software, 3, 58-66.
- National Assessment of Education Progress. (1988). Computer competence: The first national assessment. Princeton, NJ: Educational Testing Service.
- Palincsar, A. S., & Brown, A. L. (1984). Reciprocal teaching of comprehension-fostering and monitoring activities. Cognition and Instruction, 1, 117-175.
- Parnas, D. L., & Clements, P. C. (1986). A rational design process: How and why to fake it. IEEE Transactions on Software Engineering, 12, 251-257.
- Pea, R. D., & Kurland, D. M. (1984). On the cognitive effects of learning programming. New Ideas in Psychology, 2, 137-168.
- Putnam, R. T., Sleeman, D., Baxter, J. A., & Kuspa, L. K. (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: Erlbaum.

- Rist, R. S. (1986). Plans in programming: Definition, demonstration, and development. In E. Soloway & S. Iyengar (Eds.), Empirical studies of programmers (pp. 28-47). Norwood, NJ: Ablex.
- Rist, R. S. (1989). Schema creation in programming. Cognitive Science, *13*, 389-414.
- Rohwer, W. D., Jr., & Thomas, J. W. (1989). The role of autonomous problem-solving activities in learning to program. Journal of Educational Psychology, *81*, 584-593.
- Scardamalia, M., Bereiter, C., & Steinbach, R. (1984). Teachability of reflexive processes in written composition. Cognitive Science, *8*, 173-190.
- Schoenfeld, A. H. (1983). Problem solving in the mathematics curriculum: A report, recommendations, and an annotated bibliography. (M.A.A. Notes #1). Washington, DC: Mathematical Association of America.
- Schoenfeld, A. H. (1985). Mathematical problem solving. Orlando, FL: Academic Press.
- Sims-Knight, J. E. (1990). 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
- Slavin, R. E. (1987). Developmental and motivational perspectives on cooperative learning: A reconciliation. Child Development, *58*, 1161-1167.
- Swartout, W., & Balzer, R. (1982). On the inevitable intertwining of specification and implementation. Communications of the Association for Computing Machinery, *25*, 438-440.
- Trowbridge, D. (1987). An investigation of groups working at the computer. In D. E. Berger, K. Pezdek, & W. P. Banks (Eds.), Applications of cognitive psychology: Problem solving, education, and computing (pp. 47-58). Hillsdale, NJ: Erlbaum.
- Webb, N. (1984). Peer interaction and learning in cooperative small groups. Journal of Educational Psychology, *76*, 333-344.
- Wirfs-Brock, R., Wilkerson, B., & Wiener, L. (1990). Designing object-oriented software. Englewood Cliffs, NJ: Prentice-Hall.
- Wirth, N. (1971). Program development by stepwise refinement. Communications of the Association for Computing Machinery, *14*, 221-227.

Table 1 Example of Student Design

Class: MOUSE	Collaborators	
Responsibilities	Get info	Give info
Positions where to release puck from vertically Determine how sticky or slippery by holding mousebutton down Releasing the mousebutton releases the puck Is used to enter score		puck ruler puck score indicator

Class: FIELD	Collabor.	
Responsibilities	Get info	Give info
Draws itself		puck

Class: SCORE INDICATOR	Collaborators	
Responsibilities	Get info	Give info
Sends input to scoreboard		score board

Class: PUCKS	Collabor.	
Responsibilities	Get info	Give info
Draws itself, Knows where it is Deflects off wall Deflects off each other Changes color Knows how quickly to move in a horizontal plane Knows where to be released from Knows when to start moving		score board ruler mouse mouse

Class: SQUARES	Collaborators	
Responsibilities	Get info	Give info
Needs to know when the puck is inside it Draws itself Knows where it is Needs to know it is worth 1 point		score board

Class: RULER (grid)	Collabor.	
Responsibilities	Get info	Give info
Knows how sticky or slippery		

Class: DIAMONDS	Collaborators	
Responsibilities	Get info	Give info
Draws itself Knows where it is Needs to know when the puck is inside it Needs to know it is worth 10 points		score board

Class: SCOREBOARD	Collabor.	
Responsibilities	Get info	Give info
Compares score Displays score entered		squares diamonds score indicator

Table 2

Cue Cards

Identifying Responsibilities

- Make a list of verbs in description.
- Then decide which really refer to a job/function/responsibility.
- Then decide what responsibilities go with what class.
- Spend time deciding how to describe each responsibility so that it captures what needs to be done. If you don't get good descriptions now, you'll have trouble remembering what you mean.

Identifying Collaborators

Ask the following questions for each responsibility:

- Is the class capable of fulfilling this responsibility itself?
- If not, what does it need?
- From what other class can it acquire what it needs?
- Then ask to what other classes does it need to tell this information. If the information is going to be used only by the class that generates it, you don't need to put down collaborators.

Table 3
 Example of Posttest Design
 Tic Tac Toe

Class: X		Collaborators	
Responsibilities		Get info	Give info
Draws itself Knows where they are			

Class: Playing Area*		Collaborators	
Responsibilities		Get info	Give info
Draws itself Knows when 3 X's and 3 O's are in a line			

Class: O		Collaborators	
Responsibilities		Get info	Give info
Draws itself Knows where they are			

Class: Mouse Button		Collaborators	
Responsibilities		Get info	Give info
Tells X and O when to appear			