4

The Future of Computer-Assisted Design: Technological Support for Kids Building Artifacts

Mark Guzdial, Elliot Soloway, Phyllis Blumenfeld, Luke Hohmann, Ken Ewing, Iris Tabak, Kathleen Brade
University of Michigan, Ann Arbor, MI

Yasmin Kafai
Harvard University, Cambridge, MA

Abstract *A project-based learning model emphasizes student design activities to engage the student with the domain material. Students, as novice designers, need support to be successful in building artifacts. The GPCeditor (Goal-Plan-Code editor) is a computer-based design support environment (DSE) that supports high school students as they learn and do software design. Case studies suggest that students using the GPCeditor do make use of its features in learning and doing software design. However, developing solutions to the problems students still face requires addressing both technological and educational issues as part of a complex interrelationship between instructional environment, curriculum, and technology.*

INTRODUCTION

The knowledge-transmission, didactic model of American education is no longer serving the needs of students and teachers. As has been described in many studies over the last five years, students are falling behind their counterparts in other countries, even compared with other American students of years past (Farnham-Diggory, 1990).

What seems to be needed now is a more project-based (or constructionist) approach to learning. Students who are actively engaged in designing and implementing projects in a domain are also actively engaged in the learning of the knowledge in that domain. Students building artifacts are creating critiquable, sharable externalizations of their knowledge, which provides both motivation and opportunity to exercise metacognitive skills.

However, students are notoriously bad designers. For example, studies of student software designers have indicated that students do not grasp design skills such as the ability to decompose problems into modules[1] (e.g., Jefferies, Turner, Polson, and Atwood, 1981; Pea and Kurland, 1986; Spohrer and Soloway, 1985).

This research was supported by NSF Grant #MDR-9010362. Equipment for this research was donated by Apple Computer. Mark Guzdial received support from the University of Michigan School of Education, and Luke Hohmann received support from Electronic Data Systems Corporation.

Without design skills, the designer is not able to cope with the complexity of the design process or the resultant artifact (Simon, 1969). It is important for students to be successful designers because producing bad designs can remove the incentive to engage in project-based learning. Thus, students require support to do design.

The Goal-Plan-Code editor (GPCeditor) developed by the HiCE research group[2] is a computer-based design support environment[3] (DSE) that supports high school students as they learn and do software design. The purpose of the GPCeditor is to provide an environment in which students can design software (the doing of design). Over the course of the semester, students are expected to produce higher quality artifacts of ever-increasing complexity by using good, systematic design process (the learning of design).

The GPCeditor implements a set of features called *scaffolding* that work together to support the student in the design process. These supports allow the student to concentrate on the salient parts of the overall task (Brown, Bransford, Ferrara, and Campione, 1983). The GPCeditor implements scaffolding for doing design through tools and representations that encourage students to think about software as an expert does and to use a design process based on empirical studies of expert programmers. Scaffolding for learning design comes through encouraging articulation (e.g., in names and descriptions of program components) and reflection (e.g., by providing multiple, linked representations).

The GPCeditor has been in continuous use in Community High School[4] in Ann Arbor for three years. We have observed that students do produce high-quality, complex artifacts with good, systematic design process. In our process of analyzing and measuring these outcomes, we undertook a case study of four typical students in the class. For these students we gathered programs, process traces, interview data, and think-aloud protocols. This chapter describes these four students and the situation in which they worked. The following four sections cover the following topics: (a) the components of instruction used in the class, that is, the features of the GPCeditor, the instructional environment that evolved, and the curriculum used in the classroom: (b) a description of the data collected and the criteria used in their review; (c) a presentation of case studies of four students selected as a representative sample to identify benefits and problems with our approach; and (d) an outline of future strategies.

COMPONENTS OF INSTRUCTION

GPCeditor

The GPCeditor must support the students as they do design but also support the students' learning. This dual goal is accomplished by providing a rich set of design tools whose use is structured to encourage learning. For example, a library containing program fragments (whose usefulness has been proven in prior programming experience) is provided, as one would expect to provide an expert designer in order to support doing programming (Guindon and Curtis, 1988). However, in the GPCeditor, students may not use something from the library until they have first articulated (a learning strategy) a goal for which the library entry will be used.

This enforced articulation is one way in which the GPCeditor provides a specific design process. By providing a design process, we encourage learning the components of the process while the order of the processes is structured. As

students use the GPCeditor, the goal is that they internalize the process and begin using a more expert-like process resulting in the development of more expert-like products. Thus, learning occurs in a process of doing. The design process that is taught with the GPCeditor has three process stages.

Decomposition, or Analysis of the Problem

During decomposition, the student considers the problem requirements, formulates a goal, considers the potential alternative plans for achieving that goal, and finally chooses the plan for the given goal. For expert designers, the search for potential plans begins with plans already developed and used in previous plans, in the hope that the plan might be reused, thus reducing the complexity of the overall design.

The problems of student designers begin here. The skill of being able to modularly decompose problems is absolutely key in software design (Parnas, 1972) and yet is rarely developed by students even after a full semester (or more) of programming instruction (Pea and Kurland, 1986).

Composition, or Synthesis of the Solution

The student defines the particulars to make the generic plan fit the specific situation (instantiating the plan for the problem) and places the plans in the program in a particular order. Instantiating plans in software is the process of choosing appropriate data objects for the procedural plan and then ordering them in a predetermined manner to achieve a particular sequence of events.

This, too, is a key stage for student designers. A common source of error in programs, especially in student programs, is in the integration of plans (Spohrer and Soloway, 1985). Students find it difficult to take the decomposed elements and combine them into a final solution.

Debugging, or a Cycling of the Design Process

An expert programmer reviews the solution and develops predictions about the program's behavior. Testing the program, perhaps using debugging tools such as breakpoints, involves comparing expected results with actual results. If the comparison indicates that a bug exists, the programmer begins a new design cycle in which a goal may be to find the bug or perhaps to correct the bug if the cause is clear. Debugging must focus on the program as a whole to determine which plans are incorrect and then shift to concentrate on the localized interactions between component plans.

The GPCeditor provides two general sets of supports (scaffolding) in learning and doing design. The first set provides tools to aid in reflection, which is important for the learning of the goal-plan design methodology in the development of software. The second set helps students learn and do the process of design and is a critical component of the design stages described above.

Support for Reflection

The GPCeditor provides support for reflection at two levels. The first level is through the goal-plan approach, which encourages students to think about programs in a manner similar to experts. The second level supports the first by providing multiple representations for viewing the developing program.

Goal-plan approach. Students using the GPCeditor never type program state-

ments. Instead, they construct programs by defining goals (statements of what they wish to achieve) and plans (how they wish to achieve these goals) and then assembling these plans. For each goal there are usually one or more alternative plans for achieving the goal. These plans can be thought of as components, or program modules, which are composed by the student to create the complete design or program.

Plans are either defined in terms of Pascal code (e.g., a `writeln` statement is a primitive plan available to the student which writes some data to a line of the text window) or in terms of a hierarchy of subgoals and their plans. A plan that is defined through subgoals is referred to as a plan grouping. The role of data in a GPCeditor program is to instantiate a plan for a particular program. For example, the `writeln` plan is always the same from use to use and program to program, but it is made specific for a particular purpose by the choice of data to write. In this way we encourage the important design skill of tailoring program components for specific use.

The goal-plan approach encourages modular programming by making clear the difference between the what and the how. The GPCeditor takes this approach a step further and allows the students to concentrate on the goals and plans of the program without the cost of learning specific, idiosyncratic rules associated with the Pascal programming language. Students using the GPCeditor can think about their programs as plan-oriented reusable pieces, without concern for arguments, scoping, or data type incompatibilities.

For example, consider the task of writing a program:

(1) to read two input numbers from the user,
(2) add them,
(3) write the result,
(4) ask the user if there were more numbers to be added, and
(5) repeat the process if the user answers in the affirmative.

A student using the GPCeditor would not be expected to address the problem in terms of `readln`, `writeln`, and `while-do` loops. Instead, he or she is expected to think about the following:

- a `GetInput` plan (a plan grouping consisting of a `writeln` to write a prompt for the user and a `readln` for reading a value), which asks users if they would like to begin adding numbers,
- some other `GetInput` plans for reading in numbers,
- a computation plan for adding the input numbers,
- an output plan for writing the result, and
- a `DoItAgain` plan (a plan grouping containing a `while-do`, a test for equality between a string variable and a string constant such as "Yes," and a `GetInput` plan at the bottom of the loop to ask users if they would like to do the computation again).

Research on expert software designers suggests that they use such a plan-based structure when thinking about programs. Soloway and Ehrlich (1984) showed that expert designers' activity could be described in terms of a goal-plan based knowledge structure. By thinking about the design in terms of what needs to be accom-

plished (goals) and plans that meet those needs, the expert designer changes the design task from thinking about code and syntax to thinking about higher level components.

A goal-plan approach reduces design complexity by fragmenting the task and providing a framework for component reuse. A plan defined in terms of subgoals defines a level of hierarchy in the program that can be dealt with almost as a program unto itself. Thus, a program in the GPCeditor is not a monolith, but a collection of small, manageable pieces. The notion of reuse is particularly important as experts reuse pieces such as these, attempting to solve new problems by using previously generated program fragments. This reuse of plans further reduces the complexity of design by allowing the designer to treat entire branches of the decomposed task as a solved problem.

The GPCeditor supports reuse with a plan library (lower left corner of Figure 1). The student's library initially contains plans based on language constructs included in traditional Macintosh Pascal implementations. Students can add their own plans to the library (such as `GetInput` and `DoItAgain`) for later reuse. Double-clicking on a plan in the library presents information on that plan, such as its description, what the code for the plan looks like, and what data objects are needed to instantiate the plan for a program.

Multiple, linked representations. The GPCeditor provides multiple, linked representations to support students using a goal-plan approach to programming. These representations support the student in manipulating and reflecting on the goal-plan structure. Multiple representations provide opportunity to consider the problem from more than one perspective (Larkin and Simon, 1987).

Figure 1 is a sample screen from the GPCeditor. The upper left window is the goal-plan list (or *bucket,* in the students' and researchers' common language), which contains lists of goals and plans. These identify, by name, each goal and its corresponding plan at a particular level of hierarchy (named above the lists). In the example, the named goals and plans implement the `initial_questions` plan. The upper right window presents the traditional code-oriented view of the program. The lower right window gives a graphical *overview* of the hierarchy of goals and plans. Each goal appears with its corresponding plan, and if the plan is defined in terms of lower level goals and plans, those appear below and connected to the goal-plan node.

These representations are linked such that corresponding elements in all the representations appear highlighted at the same time. Clicking on a goal name in the goal list highlights the corresponding plan in the plan list, the corresponding code in the *code view,* and the corresponding node in the overview representation. The purpose for the linking is to encourage students to ease the transition between working with one representation and working with several.

Support for Process

The GPCeditor enforces the model of design described earlier in this section: decomposition of the problem, composition of the solution elements into a whole, and debugging or testing of that whole solution. This model is similar to others developed for programmers (Adelson and Soloway, 1984; Spohrer, 1989) and has elements like those used to describe other design domains (e.g., Hayes and Flower, 1980, for composition). Though fixed, it is a reasonable model of expert-like design. Whereas design models of experts emphasize that the ordering of

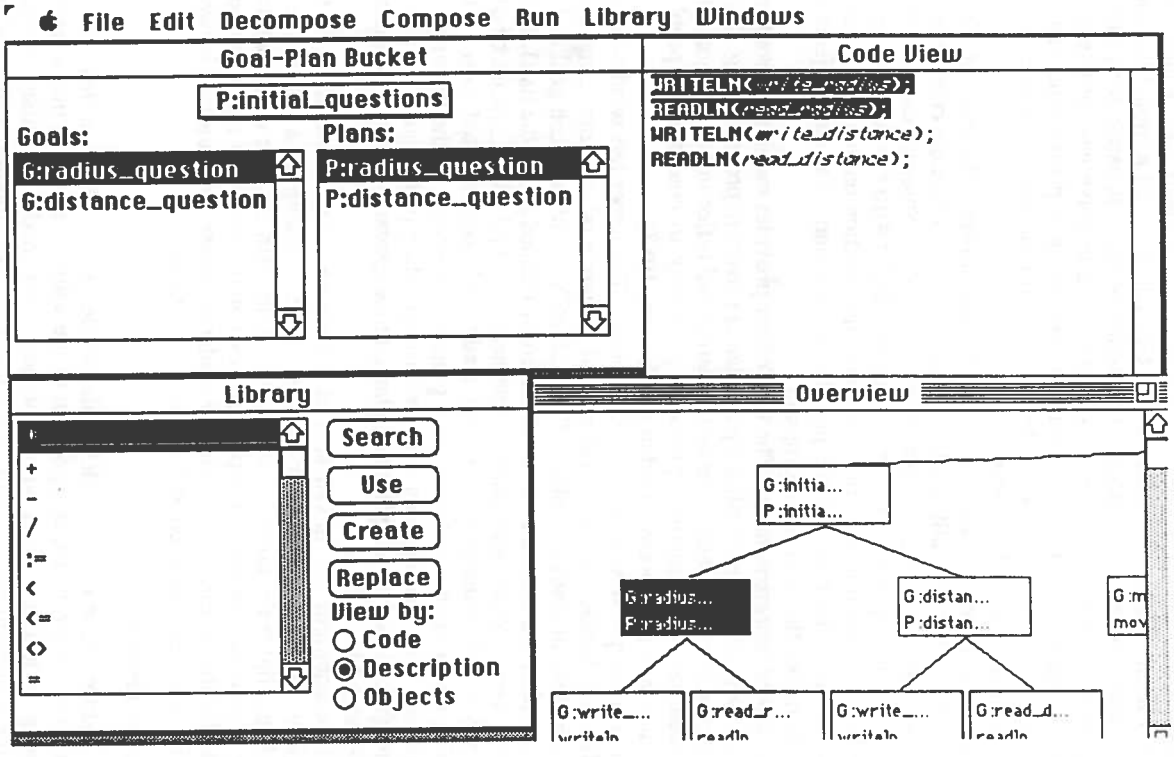


Figure 1 GPCeditor sample screen.

processes is not fixed (Hayes and Flower, 1980; Soloway and Ehrlich, 1984), students need a particular, initial ordering upon which to build their own design strategies (Corno and Snow, 1986).

The GPCeditor features scaffolding to support each of the three processes in its design model. The operations in this model are made explicit in the menus available to the student. Operations the student wishes to perform during decomposition are in the Decompose menu, and composition operations are in the Compose menu (Figure 2). Menus and their associated operations are enabled and disabled to reflect the current stage of the student's design.

Decomposition. During decomposition, the student uses New Goal . . . to formulate a goal, that is, to name and describe a goal. The library can be browsed at any time to identify potential alternative plans. The student may not select New Plan . . . to identify a plan for the program until a matching goal has been created. This means that students may not put components into their design without first identifying why the component is useful. The student can choose to create the new plan as a plan grouping or as a plan from the library by clicking on a plan in the library, then clicking on the Use button (Figure 1). As goals and plans are created, they appear in the goal-plan bucket and the overview window.

Some variation on this decomposition process does occur. For example, students sometimes use a *library-driven decomposition* strategy. They will browse through the library, check descriptions of various plans, and than create a goal and use one of the plans identified during the browse. Another variation that is not stressed within the curriculum is that a student can create any number of high-level, abstract goals that are not based on any particular order of composition. Only when plans are associated with these goals does the actual composition process occur. The important focus of the decomposition process of the GPCeditor is that goals are created before plans are chosen, with the implicit intent being the reasoned creation of these goals and articulations for the selections of a particular plan.

Composition. Once a plan is identified, the student enters the composition pro-

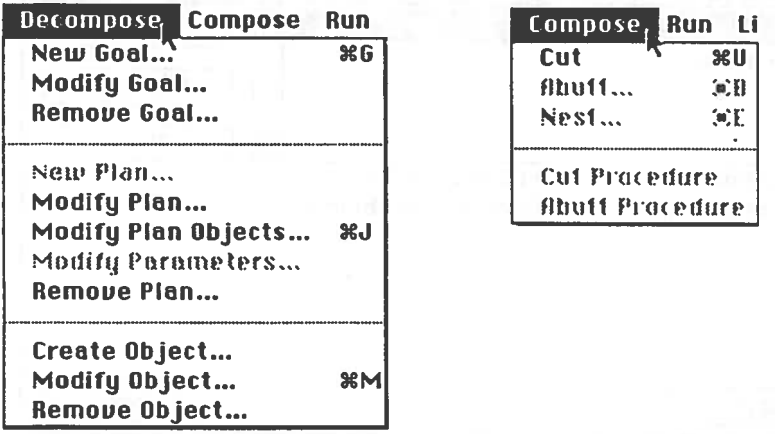


Figure 2 Decompose and compose menus.

cess. A plan just copied from the library is immediately instantiated for this program. A *match window* appears (Figure 3), listing the data objects needed to instantiate this plan for a program. From this window, the student can match each needed data object to an object already existing in the program or can create a new data object for the plan. Some required data objects might be left unmatched, to later be filled with plans consisting of expressions.

Students compose instantiated plans into the program using the Abutt and Nest operations in the Compose menu. The choice of operation depends on the kind of plan ordering desired. *Abutt* will place a plan either before or after another plan. *Nest* will place a plan within another plan, as when placing a plan between a `while-do` loop's `begin-end` block. The other compose operation, *Cut*, removes a plan from the program.

Debugging. Students change their programs during debugging by using the same metaphors used in constructing the program. Goals and plans can be removed or modified (i.e., changing names or descriptions). Plans can be matched to different data objects. However, the enforced ordering still prevails: A goal must be created before a plan can be identified, the plan must then be instantiated, and only then can the plan be composed into the program. In addition, program review tools are provided such as *Step* (see step through a program plan-by-plan to review execution) and an observe window to check variable values while the program executes.

Instructional Environment

The GPCeditor is used in Community High School, an alternative school in the Ann Arbor school district. The high school is loosely structured and emphasizes individual creativity and interdisciplinary efforts. Students do not have homerooms: They have forums that are as likely to meet in the evening at an area

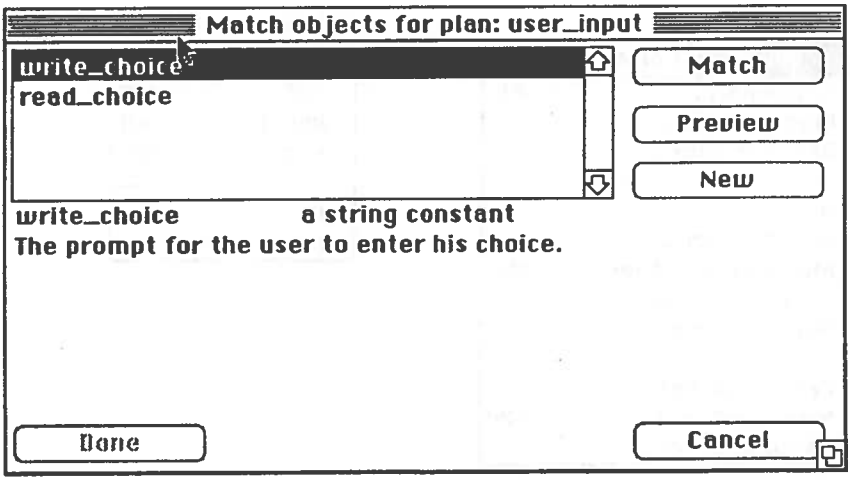


Figure 3 GPCeditor match window for instantiating plans for a program with data.

theater as in a formal classroom. Students are neither necessarily gifted nor learning disabled. They elect to come to this school because they are uncomfortable with more traditional schools.

Students apply to enter the course. Selection of students for the course is made by the administration, the course instructor, and members of the Highly Interactive Computing Environments (HiCE) research group. The goal is to fill the class with a wide range of students on dimensions such as gender, race, past computer experience, and academic performance. Ten to eleven students take the class each semester. The class using the GPCeditor has been run every semester since winter 1989. The students discussed in this chapter used the GPCeditor in fall 1989.

In the semester during which data were gathered for this chapter, the class met five days a week for two hours a day.[5] Because a two-hour class was unusual for Community High, the class was scheduled for the last hour of the normal day and one additional hour after school.

The instructor for the course, Robert Kinel, has been teaching mathematics (algebra, geometry, and calculus) for more than ten years, and computer science (specifically, Pascal) for seven years. He, with members of the HiCE group,[6] evolved the style of classroom interaction during the first semester of the course. Each student progresses through a set of worksheets at his or her own pace. The instructors wander through the classroom, providing individual instruction when asked and offering advice or asking students about their projects to encourage reflection. Thus, the instructor takes the role of the mentor, with the GPCeditor providing the needed technological support for the mentoring process. Occasionally, the class gathers to discuss a common issue.

For example, a common group discussion topic is on what makes a good design. The instructor presents students' programs in their overview representation on an overhead projector and has the students review and critique each design. The instructor might ask the students if the depth of decomposition and branching was appropriate, of if a different decomposition would have been better.

Although the course may sound teacher-intensive, in actual practice it is not. Students work alone or interact with their peers, for the most part. The teacher may provide coaching and instruction to an individual student for an entire class period without disturbing the activities of other students in the class. The GPCeditor provides enough support for many students in the class. However, even the amount of one-on-one interaction that is required might be too much for some classrooms. We address the point of needing to capture more of the teacher's expertise in the environment in the later section on future enhancements.

After approximately twelve weeks of the sixteen-week semester, some students will have progressed beyond the capabilities of the GPCeditor and begun using Lightspeed Pascal instead of the GPCeditor. Most of the students (50% to 75%) finish the portions of the curriculum that can be completed on the GPCeditor. They then use a traditional Pascal environment on the Macintosh to use features such as procedures and arrays that were not yet implemented in the GPCeditor at the time of this study.[7]

Curriculum

The goal of the curriculum is to teach students to develop Pascal software by using good design skills in the framework of the expert-based software design

process described earlier. The curriculum is organized toward developing increasingly complex programs throughout the semester while simultaneously learning the design skills necessary to cope with this complexity. Design skills are introduced in stages as students learn more Pascal and are asked to synthesize ever more complex programs.

The curriculum used in the GPCeditor class was originally developed by Jeannine Pinto and Yasmin Kafai at Yale University for use in a course at a New Haven high school. Soloway and his group taught design using this curriculum with a traditional Pascal programming environment. Without support for the concepts in the environment, they found that students continued to focus on the code and not on the goal-plan structure of design. This curriculum was updated by Kafai for use with the GPCeditor.

The course is worksheet-based and self-paced. Students complete worksheets that discuss topics in Pascal and design. They file their completed worksheets and programs in individual notebooks, which the instructor collects, comments on, and grades every two to four weeks. Table 1 lists the GPCeditor worksheets. Each worksheet has a particular focus, as indicated by Table 1. The focuses were between Pascal constructs (e.g., the `while-do` loop and Macintosh graphics functions) and software design (e.g., identification of plans and the purpose of hierarchical decomposition). There are twenty-seven worksheets and forty-one assignments in the semester.

The assignments focus on Macintosh-style graphics and user interfaces. Students develop programs to move objects around on the screen, draw faces with blinking eyes, play games, and accept input through the keyboard or the mouse.

DATA AND CRITERIA

There were three foci that we used in considering the data gathered on the four case study students: student characteristics, learning and doing design, and use of the GPCeditor.

- The student characteristics of most concern were prior experience with computers, and motivation (both at the beginning and throughout the semester).
- The issue of learning and doing design breaks into four parts: quality of overall process and quality in each of the process stages.
  - — In terms of the overall process, the use of reflection and explicit planning is key. Reflection is key to design (Jefferies et al., 1981) and to learning (Brown et al., 1983). Explicit planning is a measure of how systematic the student is about design.
  - — The other three parts of learning and doing design are the three process stages: decomposition/reflection on the whole task, composition/reflection on the ordering of components, and debugging/reflection on the whole program with integrated components. Besides looking at performance on each of these processes, a subprocess for both decomposition and composition was identified for emphasis. For decomposition, how the student began the program was seen as significant, and for composition, the student's ability to handle data.
- To study use of the GPCeditor, two components were emphasized: the library and the alternative representations. Besides being the most significant physical

Table 1 GPCeditor worksheets and their foci

Worksheet	Focus
Planning, programming, and problem solving	Introduction to the GPCeditor. Students are asked to compose existing library plans and execute their programs.
Values and variables: Viva la differences	Students learn the difference between values and variables. They construct simple programs for performing math operations.
User friendliness: The key to popular programs	Introduction to the `GetInput` plan (prompt the user for data entry and input of data).
Getting coordinated: An introduction to the GPCeditor coordinate system	Students learn to specify locations in the graphics window in terms of horizontal and vertical offsets.
Graphics in the GPCeditor	Introduction to Macintosh graphics primitives, such as `paintoval` and `framerect`.
Looping in the GPCeditor	Introduction to the `while-do loop`.
Moving pictures in the GPCeditor	Combining iteration with graphics to achieve animation.
Writing good programs	Identification of named plans such as `GetInput` and `DoItAgain`.
Plans for moving pictures	Descriptions of plans used in animation.
Problem simplification	Discussion on program decomposition.
The word of the day: Hierarchy	Identifying program hierarchy.
Line drawings	Introduction to `moveto` and `lineto` graphics primitives.
Blinking eye face	Draw a face with animated, blinking eyes—a complex program requiring 20–50 lines of program code.
Decisions, decisions, decisions	Introduction to the `if-then`.
Apply your knowledge: Identifying plans	Identifying plans in other students' programs.
Finding out what you know about plans	Quiz on plans.
Working with plans	Introduction to debugging techniques.
Plan library	List of all programming plans met in the course.
Random numbers	Introduction to `random`.
Random numbers and game programs	Creation of computer games using `random` and `if-then`.
User-friendly interfaces	Using mouse-driven input.
Procedures	Introduction to procedures.
My life in disorganized crime	Methods for organizing code (procedures).
Or: Logical alternatives	Using logical conjunction.
Procedures revisited	More on procedures.
Handling user input errors	Handling data verification.
What you do when everyone wants to play: Arrays	Introduction to arrays.

features of the environment (Figure 1), they are the workspaces in which the students design.

Table 2 summarizes the data collected in the case study. The primary data source was the think-aloud protocol. These were used to observe the students' design processes and their use of the GPCeditor. The secondary data sources were questionnaires, interviews, review of the students' assigned programs, and trace files of GPCeditor usage (which are automatically generated by the GPCeditor).

Table 2 Summary of data sources and criteria

Topic/Issue	Data source	Frequency	No. of students	Criteria
		Student characteristics		
Prior experience with programming and computers, and academic performance	Questionnaire	Twice (at beginning and end of semester)	All students	Asked students to describe previous computer experiences, programming instruction, and to write a program
Motivation	Interview	Twice (once at the beginning of the semester, and again mid-semester)	All students in initial interview, only students not in protocol in midsemester interview	Asked students why they were taking this class and what they thought of computers and the GPCeditor in particular
		Learning and doing design		
Occurrence of reflection overall	Think-aloud protocols	Three times	Four students	Earlier and more frequent reflection as more expert-like
Occurrence of explicit planning	Think-aloud protocols	Three times	Four students	Explicit statement of plans or use of external aids (notebooks, pencil-and-paper)
Ability to begin program	Think-aloud protocols	Three times	Four students	Use of heuristic to determine first goal
Able to manage complexity in decomposition/reflection on whole task	Think-aloud protocols	Three times	Four students	Greater use of plan groupings to create levels of hierarchy; use of plan groupings to create functional decomposition; reuse of plans
	Assigned programs	Daily	All students	Reuse of plans

(*Table continues on next page*)

Ability to manage complexity in composition/reflection on components	Think-aloud protocols	Three times	Four students	Tailoring of reused plans; use of composition features; correctness of program
	Assigned programs	Daily	All students	Tailoring of reused plans; correctness of program
Ability to instantiate plans with data	Think-aloud protocols	Three times	Four students	Ability to recall data objects by function
	Assigned programs	Daily	All students	Naming of variables by function
Ability to manage complexity in debugging/reflection on whole program	Think-aloud protocols	Three times	Four students	Use of review tools; skill in debugging—focus on goal and purposeful corrections
	Usage trace files	Daily	All students	Amount of program testing
		Use of GPCeditor features		
Use of the library	Think-aloud protocols	Three times	Four students	Reuse of plans; styles of library browsing
	Assigned programs	Daily	All students	Reuse of plans
	Usage trace files	Daily	All students	Reuse of plans; amount of plan saving
Use of alternative representations	Think-aloud protocols	Three times	Four students	Active reference to representations, i.e., use of the overview, goal-plan lists, and text view

Not all types of data were collected for all kinds of students, to reduce the complexity of review of the data and development of these criteria.

Each protocol had a similar structure. Students were asked to complete a series of small programs similar to ones they were doing in class. The programs were organized such that plans created in the early programs would be useful in the later ones, to encourage reuse. The worksheet from the first protocol appears in Appendix 1 as an example. In this protocol, the students were asked to write four programs: to draw a face, to draw two faces, to ask the user his name and greet him, and finally, both to draw two faces and to greet the user.

In addition to these formal data, less formal data were used to fill out the picture of the classroom. A researcher visited the class two to three times a week and kept a journal. Each student kept a daily journal on their problems, successes, comments on the GPCeditor, and comments on the entire class. Finally, the instructor's comments were often solicited.

Table 2 also summarizes the criteria used in evaluating student performance in each of these study foci using these data.

- For student characteristics, students were asked in questionnaires and in interviews what they felt about computers and the GPCeditor and what their prior programming experience had been.
- Criteria were developed for each of the subfoci to evaluate how the student did design. The data were compared over time by using these criteria to determine how well students learned design.
  - Reflection in the overall process was evaluated on the frequency of reflection and where in the process it occurred. Reflecting late in the process was inefficient because decisions had already been made earlier in the process. Explicit planning was noted by verbal planning comments and use of notes.
  - Decompositions were evaluated on the breadth and depth of the goal-plan trees, the use of plan groupings to create levels of hierarchy, and reuse of plans.
  - Compositions were evaluated on the quality of the program, the student's ability to order the plans, ability to tailor reused plans, and ability to find and use data objects.
  - Debugging was evaluated on kind and purposefulness of changes and use of debugging tools.
- Student use of the GPCeditor was evaluated on use of the library for saving and browsing and on use of the alternative representations for reflection or manipulation.

SUMMARY OF CASE STUDIES

Summary across Students

The four case study students are Sue, Allen, Lois, and Fred.[8] The data on these students is presented in detail in the Appendix 2 and are summarized in Table 3. These students were selected as a representative sample of the types of students using the GPCeditor. These students cover a wide range of prior knowledge and interests: from initial inexperience with programming to being self-taught in several languages, and from entering the class excited by computers to entering the

Table 3 Summary of data for case study students

Issues	Allen	Fred	Lois	Sue
		Student characteristics		
Prior experience	3.0 GPA, sophomore; less than a semester of Basic	Freshman; self-taught programming experience in Basic, Pascal, and Grasp	3.7 GPA, senior; self-taught programming experience in Basic	3.2 GPA, junior; less than a semester of Basic
Motivation	Interested in computer at initial interview	Interested in computers at initial interview; didn't like the GPCeditor at midsemester interview	Took class for transcript; experiences with computers, losing data at initial interview; didn't like the GPCeditor at midsemester interview	Took class for transcript; found computers frustrating at initial interview
		Learning and doing design		
Reflection overall	Early and often in first two protocols; only at beginning in last	Early and often in first protocol, only at beginning in last	Early and often in protocol	Late and little reflection in all three protocols
Explicit planning	Used notes in first protocol, and discussed where he was going; made some explicit plans in second protocol, few in third	Made explicit verbal and written plans in first protocol; none in last	Used notebook and made verbal plans in protocol	Little planning
Starting program	Had no problem starting in any of the three	Had no problem starting	Used a heuristic of doing something and then building on that	Frequently didn't know where to begin
Decomposition/reflection on whole task	Used composition to functionally decompose programs in the first protocol; only used groupings for difficult problems in second protocol; decomposition was not observable in third; no reuse of plans	No use of plan groupings; no reuse of plans	Reuse of plans observable in assigned programs; use of good functional decomposition	Little use of hierarchical decomposition in protocols, and only in rote manner; assigned programs had good functional decomposition and reuse, eventually

(*Table continues on next page*)

Table 3 Summary of data for case study students (*continued*)

Issues	Allen	Fred	Lois	Sue
Composition/reflection on components	No tailoring of reused plans; no problems composing plans; programs functioned correctly	No tailoring of reused plans; few problems composing plans in first program (choosing between *abutt* and *nest*); programs exceeded requirements	Tailored reused plans; programs functioned correctly	Tailoring of reused plans: programs exceeded requirements, eventually
Data handling	Poorly named data objects, but no problems finding data	Poorly named data objects; problems finding data in first protocol; no problems in later programs	Well-named data objects with no trouble finding data objects	Well-named data objects: many problems finding data in first protocol, fewer in later protocols
Debugging/reflection on whole program	Used hierarchy as a debugging technique; little debugging in later protocols	Used output and code view for simulation when debugging. Lost track of path in first protocol. Successful debugger later	Used simulation and tools	No identifiable debugging strategy; made random changes
		Use of GPCeditor features		
Library use	Little use of saving, reuse, or browsing	Extensive browsing in first week; no saving or reuse	Little use until latter part of semester; saving and reuse	Frequent browsing throughout semester; saving and reuse
Alternative representations	Used code view, goal-plan buckets, and overview from first protocol	Used code view only in first protocol; occasional use of goal-plan buckets for plan selection in second protocol; never referenced the overview	Little active use of alternative representations	Used code view primarily, with occasional use of the goal-plan buckets

Table 4 Data collected for case study students

	Think-aloud protocols			Interviews	
Student	First	Second	Third[1]	Initial	Mid-Semester *r*
Sue	X	X	X	X	
Allen	X	X	X	X	
Lois			X	X	X
Fred	X		X	X	X

[1]Third protocol was with Lightspeed Pascal for all students except Sue.

class frustrated by computers. Their performance varied widely: On the first protocol, Allen completed all four programs, Fred completed the first two, and Sue finished only the first. The learning and doing of these students offers an representative sample of the kinds of activity seen when using the GPCeditor.

As mentioned in the previous section, not all data were collected for all students. The specific data collected for these four students are summarized in Table 4. The most data are available for Sue and Allen, the least for Lois.

In general, the students in the GPCeditor class seemed to succeed well using it. Table 5 describes the case study students' final projects by using a rough measure of program complexity, the number of lines of code in the program. For a first-semester high school programming course, these are large programs for students to make functional, especially considering how students in traditional courses rarely get beyond syntactic correctness in their programs (Pea and Kurland, 1986). Individually, the case study students point out the strengths and weaknesses of the GPCeditor approach.

Sue

Sue did seem to grasp the design concepts being taught with the GPCeditor. Sue used hierarchical decomposition, she reused plans, and she wrote working programs. The downside was that her progress was labor-intensive both for her and the instructor.

She needed more low-level support than the GPCeditor provided. For example, she might have found the GPCeditor more useful if it provided heuristics for beginning a program, more explicit process support (e.g., a prompt suggesting "Stop here and write down all the possible goals you might use"), and coaching (e.g., a phrase like "For this program, you will probably find `moveto` to be very useful").

Table 5 Size of final projects in number of lines

Student	Size	Environment
Sue	41	GPCeditor
Allen[1]	33 and 88	Lightspeed
Lois	147	Lightspeed
Fred	144	Lightspeed

[1]Allen did two programs for his final project.

Allen

Allen began using the GPCeditor features and design skills it exemplifies, but then returned to novice-like skills. Though he continued to produce significant programs, they were ill-structured and he did not use the tools of the environment. He often complained that it was easier to work without the tools and that the GPCeditor slowed him down.

For Allen, the GPCeditor would have served him better if it provided more high-level support. The tools of the GPCeditor are well-designed for novice students, but as students became more expert the tools become more of a nuisance than an aid. Allen might have appreciated tools for structuring programs and for tracking data that are more like those appearing in expert-level CASE (computer-aided software engineering) tools.

Lois

For Lois, motivation was key. She was not successful in the course until she encountered problems that captured her interest. Once she had latched on to those, she used the functionality of the GPCeditor and learned the design skills being taught. By the end of the semester, she was one of the most proficient designers in the class.

Fred

Fred was an accomplished programmer entering the class. He did learn design skills in the course, but he did not explicitly use the ones being taught. From his comments, he may not have recognized the usefulness of the GPCeditor tools and design skills.

Fred might have found useful some instruction that explicitly modeled the design process. As design processes are dynamic, they are difficult to transmit through worksheets and thus require the instructor to model the design process (Collins, 1988). By performing the task, using the tools, and explicitly demonstrating good process, the instructor can provide a model for the students to follow (Paris and Winograd, 1989).

Instruction in the GPCeditor class did not include expert designers using the tools of the GPCeditor. Perhaps if Fred might have seen how the overview could be used effectively, where decomposition was useful, and what good debugging strategies were, he might have practiced them.

Summary across Study Foci

Though four students are too small a sample to make any statements of significance, the case studies themselves can be summarized across study foci. These provide some indication of how students are described under each of these foci.

Student Characteristics

Students' past experiences with programming did not seem to affect their performance with the GPCeditor as much as did other factors. Students with little experience were able to design complex programs with high quality. Students with a lot of experience generally did try the GPCeditor and used it in ways similar to other students. For example, Fred had significant programming experience before entering the class, but he had never done the kind of programming before (e.g.,

the size and complexity of the programs) that he did in the GPCeditor class. From his comments and performance, he seemed to be giving the GPCeditor a chance.

The students' motivation seemed to be more significant than previous programming experience. Neither Lois nor Sue were particularly interested in the course or the GPCeditor at the beginning of the semester. Neither performed well during the first few weeks. Lois only began to improve when she grew interested in the programs she was writing. Sue did not show much improvement in the class nor interest in the programming assignments.

Learning and Doing Design

In this section we examine four distinct parts: the overall process and the process within the three stages of decomposition, composition, and debugging. The data indicate that even with the scaffolding of the software design process provided in the GPCeditor, our students still exhibited most of the problems associated with a novice-like approach to software design.

(1) *The overall process.* The students' overall process was not very good. Though several of the students began with good explicit planning and early and frequent reflection, these characteristics faded in the later protocols. Those students who did plan and reflect on the process performed well. Those who did not produced decompositions of a poorer quality relative to other students. In some cases the decompositions were unintelligible, although this was rare.

(2) *Decomposition.* The students' decomposition were, in general, quite good, and they did seem to improve over the course of the semester. Considering that students are notoriously bad at modular decomposition (Spohrer and Soloway, 1985), the quality and use of plan groupings to create hierarchy in their decompositions was impressive. Most students had good heuristics for handling problems such as determining where to begin a program. However, there was little reuse of grouped plans.

(3) *Composition.* The students' compositions were less impressive. Though their programs did run, data names were poor and the meaning of the data object associated with the variable name was often forgotten. The compositions were poorly structured and hard to read. For example, Allen's example program described in Appendix 2 has assignment statements (whose purpose is unclear because of variable names such as `A` and `vari`) interspersed among supposedly identical loops. Sue's composition, though much clearer and better structured, includes redundant plans not related to the problem at hand as well as data objects that are never used. The compositions did not seem to improve during the semester.

(4) *Debugging.* The students' debugging was interesting in its diversity. The students' use of debugging tools ranged from examination of the hierarchy as a debugging tool to use of mental simulation. Both Fred and Sue had significant difficulty keeping track of program bugs and following up on their correction. Students did develop debugging strategies across the semester.

The results are consistent with the previous claim that American students have more experience in the analysis skills of decomposition and less experience with the synthesis skills of composition and debugging.

Use of GPCeditor Features

In this section we summarize interactions of the students with various features in the GPCeditor. These observations motivate and direct changes to the existing environment and modifications to the course curriculum.

(1) *The Plan Library.* The plan library was mostly used for browsing, for finding information about plans. Though both Sue and Lois did use the library for saving, reusing, and tailoring plans (later in the semester), most students did not. Neither Fred nor Allen made any use of the library other than for retrieving plans and some browsing. The library use, however, did change during the course. Sue began saving and reusing plans early, but Lois did not until later. Fred began the semester doing a lot of browsing, but then he stopped browsing.

(2) *Goal-Plan Lists, Code View, and the Overview.* Most students used the goal-plan list (buckets) and the code view for reflection and manipulation. It is notable that they did not use just one, but it is surprising that they did not use the overview. Allen was unusual in his use of the overview, because most students were like Fred, who found it useless. Many students covered up the overview with other windows and never uncovered it.

(3) *Debugging and Other Advanced Features of the GPCeditor.* There was little use of any of the advanced debugging features of the GPCeditor. The Step and Walk program run options were rarely used, and students did not interact much with the observe window. The GPCeditor does provide the ability to set breakpoints (e.g., temporarily suspend the execution of a program so that the values of internal data objects can be easily examined or modified) but this feature was not used.

Other facilities provided by the GPCeditor as tools to support the design process were rarely used. For example, the plan library provides a search capability, allowing the student to search the plan library for all plans that contain a key-word phrase in their plan description.

FUTURE DIRECTIONS AND SUMMARY

Changes to the Instructional Environment, Curriculum, and GPCeditor

The problems described by the case study analyses are not answered by changing any one portion of the instructional package used in the GPCeditor class. The pieces of the package interact and require changes to all three components to be effective.

Changes to the Instructional Environment

The GPCeditor supports students as they learn and do software design, but our students had trouble learning which features of the GPCeditor should be used in various stages of the design process. Furthermore, it is unlikely that students will spontaneously discover certain key aspects of the design process (such as advanced debugging strategies or the use of alternative designs) without explicit instruction. The question is how to demonstrate the process of software design used by expert programmers that forms the foundation of the GPCeditor.

The question can be addressed most effectively by having the instructor model the use of the GPCeditor by solving problems similar to those given to the students. During this modeling the instructor would be expected to make explicit the rationale for using specific features of the GPCeditor. This would enable students to better realize how the instructor's actions in solving problems can be brought to bear on the problems they are attempting to solve.

Changes to the Curriculum.

The current focus of the GPCeditor curriculum is based on the analytical skills of decomposition and the synthesis skills of composition. Though the goal-plan approach is covered in a strong fashion, the curriculum does not talk about reflection, heuristics, and why the GPCeditor tools should be used. The curriculum needs to cover this material to enable learning of these design concepts.

The assigned programming tasks in the curriculum are not intrinsically motivating. They were designed to teach students to create the sorts of effects and interfaces that users encountered on the Macintosh. The assumption was that students would be interested in creating programs like those that they were using. Although this was effective for some students, it was not especially motivating for others. More motivating tasks might be those grounded in real-world problems (Collins, 1988; Harel and Papert, 1990).

Changes to the GPCeditor

The GPCeditor currently provides reasonable support to one type of student. It needs to change to provide a wider range of support for individual students (both low- and high-ability students) and to enhance the instruction and the curriculum.

A needed enhancement to the GPCeditor is adaptable scaffolding. The existing scaffolding of the GPCeditor, as exemplified by the strict process control of the student in the design process, is fixed. When first using the GPCeditor, the strict process control provides a structure that enables the student to handle the complexities of the design. As the student's skill grows and they become more expert-like in their problem-solving process, the strict process control can impede the student. The scaffolding of the GPCeditor needs to fade in such a way as to provide a less strict framework for the solution of problems.

Additional support the students might find helpful includes:

- Prompts that could be added to the GPCeditor to provide more explicit support for process learning. Example prompts might inform students when they should reflect or suggest what they might be thinking about for effective problem solving (Polya, 1945).
- Suggestions by the GPCeditor of plans, strategies, and heuristics, if it were aware of the kind of program being worked on. This kind of task-specific support might be useful as low-level scaffolding.
- New representations that could be added to insure complete coverage of the design process. In particular, tools could be created to ease the process of tracking data objects and of recalling past activity.

We do not anticipate that the implementation of adaptable scaffolding and advanced design support will be based solely on students' interactions with the GPCeditor. Rather, we expect to provide tools within the GPCeditor that allow the

student with the advice of the instructor to customize the environment for a given student. Ultimately, we plan to provide a set of customization tools to the students themselves for more complete control over their own design process.

Summary

The GPCeditor has shown the validity of using computers as design support environments (DSEs) for students. The support provided in the GPCeditor has enabled the students to go beyond what they might achieve in a typical programming environment, to develop complex and interesting artifacts that motivate them and provide a focus for their learning.

We see DSEs as being the next stage in the evolution of computer-aided design (CAD) environments. Students are just one kind of novice engaged in design. The computer has made many design domains available to users, such as publishing and even architectural design. These domains are now available because the computer has taken over the mechanical skills previously necessary to work in that domain. However, design in these domains is more than simply mechanics—experts in these domains have knowledge structures and skills that make them capable of working on complex artifacts. For design novices to be at all successful in designing in these new domains, we must attempt to provide some of this expert knowledge in the form of design support in the environment.

The GPCeditor has provided useful information in our first pass at providing DSEs for students. Clearly, additional factors of instruction, curriculum, and the interaction with the environment must be considered and used to enhance the final result. Nevertheless, the direction is also clear that providing support for student design activities is key to making project-based learning a reality in the classroom.

ACKNOWLEDGMENTS

The GPCeditor was designed by Ken Ewing at Yale University. A team of programmers completed implementation at the University of Michigan: Luke Hohmann, Dave Koziol, Dan O'Leary, Charles Weaver, and Mark Guzdial. Luke Hohmann has maintained and updated the environment for the last three years.

REFERENCES

Adelson, B., and E. Soloway, "A cognitive model of software design," Technical Report #342, Cognition and Programming Project. Yale University, New Haven, CT, 1984.

Brown, A. L., J. D. Bransford, R. A. Ferrara, and J. C. Campione, "Learning, remembering, and understanding," in W. Kessen (ed.), *Handbook of child psychology: Cognitive development*, 77–166. New York: Wiley, 1983.

Collins, A., "Cognitive apprenticeship and instructional technology," Technical Report #6899. Cambridge, MA: Bolt, Beranek, Newman, 1988.

Corno, L., and R. Snow, "Adapting teaching to individual differences among learners," in M. Wittrock, *Handbook of research on teaching*, pp. 605–629. New York: Macmillan, 1986.

Farnham-Diggory, S., *Schooling*. Cambridge, MA: Harvard University Press, 1990.

Guindon, R., and B. Curtis, "Control of cognitive processes during software design: What tools are needed?" in *CHI'88: Conference Proceedings: Special Issue of the ACM/SIGCHI Bulletin*, 263–268, 1988.

Harel, I. *Software design for learning: Children's construction of meaning for fractions and LOGO programming*, Ph.D. dissertation, MIT Media Technology Laboratory, 1988.

Hayes, J. R., and L. S. Flower. "Identifying the organization of writing processes." In L. W. Gregg and E. R. Steinberg (eds.), *Cognitive processes in writing*, Hillsdale, NJ: Erlbaum, 1980.

Jefferies, R., A. A. Turner, P. G. Polson, and M. E. Atwood, "The processes involved in designing software," in J. R. Anderson (ed.), *Cognitive skills and their acquisition*, pp. 255–283. Hillsdale, NJ: Erlbaum, 1981.

Larkin, J. H., and H. A. Simon, "Why a diagram is (sometimes) worth ten thousand words," *Cognitive Science*, 11:65–99, 1987.

Paris, S. G., and P. Winograd, "How metacognition can promote academic learning and instruction," in B. F. Jones and L. Idol (eds.), in *Dimensions of thinking and cognitive instruction*. Hillsdale, NJ: Erlbaum, 1989.

Parnas, D., "On the criteria to be used in decomposing systems into modules," *Communications of the ACM*, 15(2):1053–1058, 1972.

Pea, R. D., and D. M. Kurland, "On the cognitive effects of learning computer programming," in R. D. Pea and K. Sheingold (eds.), *Mirrors of minds*, Norwood, NJ: Ablex, 1986.

Polya, G., *How to solve it: A new aspect of mathematical method*, Princeton, NJ: Princeton University Press, 1945.

Simon, H. A. *The sciences of the artificial*, Cambridge, MA: MIT Press, 1969.

Soloway, E., and K. Ehrlich, "Empirical studies of programming knowledge," *IEEE Transactions on Software Engineering*, 10(5):595–609, 1984.

Spohrer, J. C. *MARCEL: A generate-test-and-debug (GTD) impasse/repair model of student programmers*, Ph.D. dissertation, YALEU/CSD/RR #687, Yale University, New Haven, CT, 1989.

Spohrer, J. C., and E. Soloway, "Putting it all together is hard for novice programmers," in *Proceedings of the IEEE International Conference on Systems, Man, and Cybernetics*. Tucson, AZ, pp. 728–734.

NOTES

1. The skill of defining a task in terms of subtasks is *modular decomposition*, which reduces complexity of a large task by allowing the designer to concentrate on solving smaller, more manageable component tasks. Design researchers emphasize modular decomposition as key to coping with design complexity (e.g., Parnas, 1972).
2. Highly Interactive Computing Environments Research Group, directed by Professor Elliot Soloway at the University of Michigan Electrical Engineering and Computer Science Department.
3. The GPCeditor runs on Apple Macintosh computers with at least 2.5 Mb of memory, a hard disk, and a large monitor. The version used in this study required a 19-inch monitor, but a new version can be run on 13-inch monitors.
4. Through a donation from Apple Computer, eleven stations were placed in Community High School.
5. The class now meets three days a week for ninety minutes, with optional after-school sessions.
6. During the first month or two of the first semester using the GPCeditor, three instructors were present daily (the class instructor, Soloway, and Yasmin Kafai, one of his research assistants). Once a style of interaction was established, the number of instructors dropped to two (the class instructor and a research assistant).
7. Procedures have since been implemented in the GPCeditor.
8. Student names have been changed to protect subject anonymity.

APPENDIX 1: FIRST PROTOCOL WORKSHEET

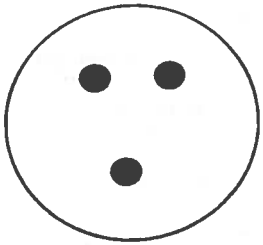
Making Faces with the GPCeditor

This worksheet is a review of concepts already visited with the GPCeditor. Here you'll put them together.

There are several components to this worksheet. **Please read over the entire worksheet, then begin at Part 1.** Do as much as you can of the worksheet.

Part I: Drawing a Face (or Maybe a Bowling Ball)

You've drawn circles, lines, ovals, and rectangles in this class, sometimes as frames and sometimes painted. If you combine a big framed circle with three smaller, painted circles (two for the eyes and one for the mouth) you can draw a face as seen below.

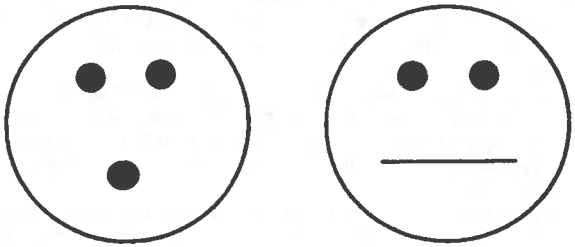


(It either looks like Mr. Bill, the clay figure from "Saturday Night Live," or a bowling ball with oddly spaced holes.)

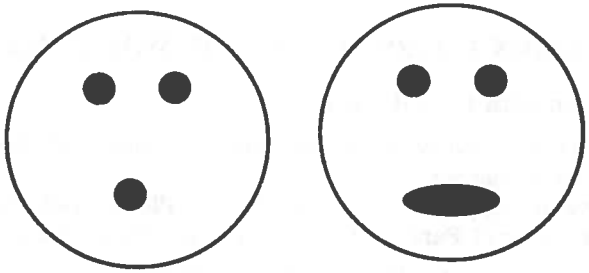
Write A Program that draws this figure, then save the program as **PartOne.**

Part II: Draw Two Faces

The mouth can be made with a circle, above, or it could be made with other objects. For example, here's the mouth made with a circle and a new one made with moveto and lineto.



Or, if you're not comfortable with moveto and lineto, you could draw the second face using paintoval.



Write A Program that draws two faces on the screen, one next to the other. The leftmost face should be the face with a paintcircle mouth. The rightmost face has a mouth drawn either with moveto and lineto or with paintoval. When you're finished, save the program as **PartTwo.**

Part III. Get the User's Name

Now you'll write an entirely new program.

Earlier in class you wrote the Echo program. This program asked the user to type something, then displayed whatever the user typed. The program looked like this:

```
program echo;
var

    echo: string;

begin
    readln (echo);
    writeln (echo);

end.
```

Write a program like Echo, but reading and writing the user's name. Write the program to ask the user for their name and read their name as input. Then, display the word "Hello" and their name (on two separate lines.)

The program says: What is your name?
You type: Mark

The program says: Hello
Mark

Part IV: Making Friendly Faces

Now, write a program that contains parts of both programs.

First ask the user for their name (in the text window).
Then draw the faces (in the graphics window).
And finally display "Hello" and their name (in the text window).

Be sure that you get the order correct: ask for the name, then draw the faces, then say hello.
You may wish to use the library to copy plans between programs.

APPENDIX 2: EVALUATION OF CASE STUDIES

Each of the four case study students (Sue, Allen, Lois, and Fred) is discussed in some detail in the following subsections. The data for each are discussed under each of the study foci.

As representative of the programs that the students produced, each student's "Challenge" program is presented. This program involved extending a program written earlier by the students called "Triangles" which drew a moiré triangle by using multiple lines in an animation in which the triangle grew across the screen horizontally. The challenge was to draw four triangles so as to create a rectangle: the original horizontal triangle to form the top, then a vertical triangle to form the right side, another horizontal for the bottom, and a final vertical triangle for the left side. The instructions for "Triangles" and "Challenge" are in Figures 4 and

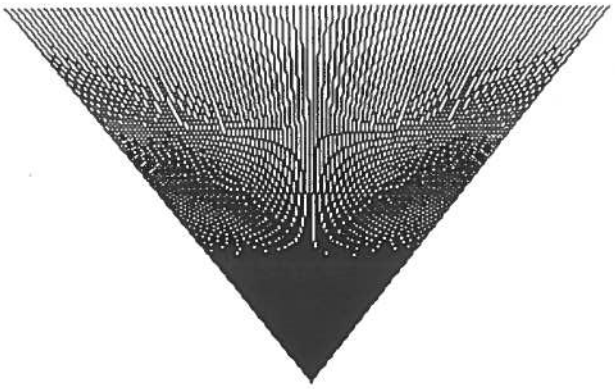


Figure 4 Instructions for Program 7: Triangles. Write a program that produces a figure like the one shown above. To do this, you will need to use both `moveto` and `lineto` and the `while` loop. Hint: Each line is drawn between a constant point and a variable point. The variable point varies only horizontally. Vary the horizontal position on this point according to a number that the user inputs when the program runs. For example, this figure was produced when a user input the number 3.

5. These programs appear in the worksheet "Moving Pictures in the GPCeditor," the seventh worksheet in the GPCeditor series of twenty-seven worksheets.

For each student, the complete Pascal program and a portion of the goal-plan decomposition is presented. The decomposition is presented as it would appear in the overview window. Plan names that begin with "P:" are plan groupings that are created by the student, and all other plans are taken from the library. Goals and plans appear left-to-right, top-to-bottom in the order in which the student created them.

Sue

Student Characteristics

Sue entered the GPCeditor class as a junior with a 3.2 GPA. She had done some programming in Basic. She admitted that she was taking the class only to have a computer class on her high school transcript. In her initial interview, she expressed frustration in her experience with computers (verbal comment by the researcher appear in italics).

Do you like computers?

"If I know what I'm doing, I like them. If I don't understand, I get frustrated."

Is that (frustration) the hardest part of using computers?

"Yeah, just, in simple terms, I don't know, trying to express the ideas that you want, and trying to get it onto the screen, trying to figure out what steps you have to do to get things onto the computer or whatever."

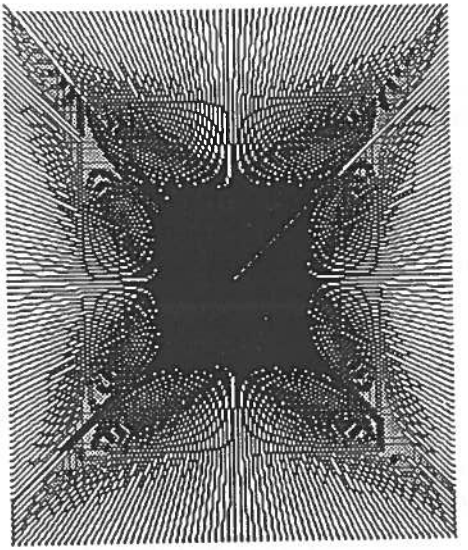


Figure 5 Instructions for Program 8: Challenge. Enhance your Triangles program to draw lines around all four sides of the screen, clockwise. See figure. Hint: You will need four `while` loops, one for drawing each side.

Use of Reflection and Explicit Planning

Sue did not reflect on the program until late in the task, when it was difficult to backtrack. For example, she would question her choice of goal or plan when instantiating the plan for the program in her protocols. This did not improve during the course of the semester.

She did not use explicit planning. She never wrote anything down, nor did she even make a verbal statement of what she was planning on doing. Not surprisingly, she often got lost in her protocols:

> "Oh! What did I do? Oh, I'll have to change that. The vertical. I forgot what I changed."

Decomposition

One of Sue's greatest difficulties was determining her first goal when faced with a new program. When given a new assignment, she would literally sit in front of her computer for an hour or more without creating any new goals or making any notes. She took longer than the other students and required extensive help. On a typical program, Sue took over three times as long as other students, ten days versus three days, and she mentioned in her journal receiving help on four of those ten days.

Part of the reason that she took so much longer in writing her programs was that she took great pains to have good hierarchical decomposition and plan reuse in her program, going so far as to redesign programs in order to achieve these goals.

The usage trace files show that she saved ten plans to the library during the course of the semester, more than twice as many as any other student in the class. The following quote from her journal exemplifies the effort that she went through for good decomposition and plan reuse.

> "I retyped the Challenge so that I could have Hierarchy and reuse some of the same plans and goals. It really wasn't that hard because I knew what I was doing but now I have to change things so that it will make sense for the next loop in the programming."

During protocols, her use of hierarchy was rote. She announced during her second protocol that she was using "plenty of hierarchy." However, she only decomposed `GetInput` plans and bodies of `while-do` loops, two purposes explicitly suggested in the worksheets. She never used a new level of hierarchy, for example, to identify a logical partition in the program.

Sue's first protocol session was marked by a lack of planning. She did not reflect on the task and her process early, but late when it was difficult to backtrack, for example, while instantiating a plan for a particular program. In particular, she lost track of data objects and her rationale for the changes she was making. She created no hierarchical decomposition. In one of the programs in the second protocol, she was asked to write the result of a calculation. She created a new level of hierarchy, placed a `writeln` within it to write the result, then placed a `readln` immediately afterward—as if she was creating a `GetInput` plan. She did not realize that one could use a `writeln` outside of a `GetInput` plan and without a `readln` following.

Composition

Sue had little trouble composing her plans, but she had significant problems matching data objects. Though she chose good, descriptive names for her data objects, she still had difficulty matching the function of the data object to the name. In her second protocol, for example, she had variables named horizontal and vertical, but she forgot how this related to the output.

> "I don't know—I'm just trying to figure out. Can I just match this? (*Pointing with her mouse to a data object.*) Left and right would be horizontal, up and down would be vertical? What should I do? I don't know."

Debugging

Her debugging strategies were typical for a novice programmer but did not improve through the semester. She made random changes that dealt only with surface characteristics of the program (e.g., changing the order of two plans) as opposed to reconstructing portions of the program (e.g., removing and creating a new goal). The following scene is from a debugging episode in her first protocol session.

> (*Sue stares at the code view, clicking on various pieces.*)
>
> "Hmm, I'm not sure . . ."
>
> (*She clicks on a plan to draw a circle, moves it one statement further down in the program—a change that has no effect on execution. She runs the program and sees no difference. She clicks on the code view again.*)
>
> "I had it right before."

(Clicks on the output graphics window.)

"Maybe it's here."

(She changes the value of a data object, then re-runs the program.)

"Okay, that did make a difference."

GPCeditor Use

Sue made more use of the library than any other student. She browsed the library and checked plan descriptions frequently. As mentioned, she saved and reused plans more than any other student.

She made little use of the alternative representations. Protocols and usage trace files showed no use of the overview and little use of the goal-plan buckets. Her primary program representation was the code view.

Sample Program

Sue's "Challenge" program and decomposition appear in Exhibit A. The program has a nice structure, is understandable, and is well composed. She uses descriptive data object names. Her program is not as efficient as it might be. In fact, she includes unnecessary components. For example, not only does she unnecessarily repeat `center := one_hundred;` within every `while-do` loop (unnecessary because `center` is never changed), the variable `center` is never used at all. The repetitive nature of the program is highlighted in the identical structure used in drawing each triangle of the challenge.

Her decomposition is modular, easily understood, and easily modified. Each triangle in the challenge is made from the same three parts: a `while-do` loop, an expression to test for ending the loop (which she names `prompt`), and a body named `while loop`. This structure is repeated through all four components, and the `while loop` plan is reused in each case. Note that the `while loop` plan is not simply reused at each case—it is also tailored to fit the different requirements of each of the loops. Though she did not place all three components in one plan grouping, she does seem to understand the basic notion of modular decomposition and plan reuse.

Allen

Student Characteristics

Allen took the GPCeditor class because he was interested in computers. He had studied Basic for six weeks when he was in sixth grade. He took the class as a sophomore with a 3.0 GPA.

Use of Reflection and Explicit Planning

In his first protocol, Allen made frequent verbal statements of what he was going to do and what he was thinking about. In these examples from the first protocol, he makes explicit verbal plans, makes use of pencil and paper to plan what he is doing, and makes use of the overview window for reflection.

"How do we do this? Get their name, then draw the faces, and finally display hello and their name."

"The vertical would be . . . Let's draw a picture for a second."

(*While clicking in the overview window*) "I always like to see what's going on over here, just to see where I am, what it looks like."

Reflection and planning were not explicit in his second and third protocols. However, he completed all of his programs correctly in both of those protocols, and in less time than anyone else. It may be that the programs were too easy for him to demand reflection or explicit planning, or that these processes were internalized and were no longer being verbalized.

Decomposition

Allen used and seemed to understand the need for hierarchical decomposition from the start. In his first protocol, he created a plan grouping and explained its purpose to the researcher.

"Let's add a little hierarchy here . . ."

Why did you create that?

"So I can put all the circles, the face part in that. And then if I put the face part in that, then I can just . . . add the different parts of the program, add those in a different part."

Allen's other protocol programs and assigned programs used strange combinations of hierarchy and ungrouped plans. In his second protocol session, programs that he began with a comment like "this is easy" invariably had little or no hierarchy (i.e., he built his program entirely out of ungrouped Pascal primitives as opposed to defining levels of hierarchy.) However, when he encountered a program bug, for example, he would often define a new level of hierarchy and decompose the difficult plan there. For Allen, hierarchy seemed to be a method of coping with a complex problem.

It may be that Allen defined a new level of hierarchy in order to try an alternative design to meet the problem at hand. Because for Allen building a plan was more easily accomplished "from scratch," previously stored plans in his mental library were constructed in the problem-solving process. However, new plans, or alternative designs, required a new level of hierarchy to deal with the complexity.

Composition

Allen never exhibited any problems with ordering and composing plans or with instantiating plans with data. This is unusual because his names for variables were obtuse and seemed to lack any connection with their function.

Debugging

Allen's debugging strategies were difficult to observe because he made few mistakes. As mentioned, he used hierarchy as a debugging technique in his second protocol. By the end of the semester, Allen's process was expert-like. He made few mistakes in his third protocol session. His usage trace files showed few program modifications and few program executions.

During the first four weeks of the semester, Allen ran his programs an average of fourteen times a day. However, during the last four weeks (when he still used the GPCeditor before going on to Lightspeed), he ran his programs only three times a day. It may be that many repeated runs early in the semester helped solidify many of the design processes, thereby reducing the need for large numbers of runs at the end of the semester.

GPCeditor Use

Allen made little use of the plan library other than access to Pascal primitives. Over the course of the entire semester, Allen's usage trace files show that he saved only one plan to the library. When asked why he never reused plans, he replied that it was "easier to build it up [from scratch]."

Allen did make extensive use of all the representations. He seemed to use the goal-plan buckets to select goals and plans for operations and to navigate the hierarchy, to use the code view for reflection during debugging, and to use the overview for reflection during decomposition.

Sample Program

Allen's program and decomposition appear in Exhibit B. His program is not as understandable as Sue's. His poor naming and duplication of data objects is immediately obvious. His variables are named `z`, `B`, `A`, and `x`. He has three constants, `vert_400`, `vert_400_b`, and `con_400`, all of whose value is 400.0. His program does not make the repetitive nature of the program clear—the first loop of his program has a distinctly different construction from the latter three.

His decomposition is nearly indecipherable, with poor naming and mixed structure. Under a plan named `inativate`, he has the `moveto` and `lineto` (graphics primitives for drawing lines) of the first loop. He groups all but the first of the `while-do` loops in one plan named `buncho loops`. The meaning of goals named `verblenumber` and `vernumB` is unclear.

Lois

Student Characteristics

Lois took the GPCeditor class as a senior (3.7 GPA) interested in studying art. She applied for the class "to get an edge on college applications and stuff." She had typed in some programs from magazines in Basic a few times. One of her first comments on computers in the interview was how a computer once lost her data.

> *Do you like computers?*
>
> "I don't really know them. I never use them. I've never programmed before . . . They organize and store stuff. They can also mess things up. I lost a bunch of stuff once. We worked for two weeks to get it back."

At the midsemester interview, she admitted that she didn't like the GPCeditor: "It's interesting. But it's not one of my favorite things."

Use of Reflection and Explicit Planning

Lois was the only student who explicitly asked for her notebook containing her worksheets during her protocol. She used her notes to identify programs and plans that were similar to one she was facing.

> "I'm looking through Moving Pictures [worksheet] and can't decide if I should take a whole program or part. COMET, SUPERNOVA [names of programs in that worksheet], it does increase the radius, doesn't it? I don't want to change the size of the square . . . Okay, I'll have to use a moveto."

Lois reflected early and often in the design process. She often paused to think about her program and how she was planning to complete it.

Decomposition

At the beginning of the semester, Lois had a hard time decomposing tasks and choosing the plans for the goals that she had chosen. Her comments in her diary, and the instructor's comments to the researcher, suggest that she received personal instruction almost daily. However, by midsemester, she appreciated and used hierarchical decomposition in her programs. In her midsemester interview, she explained why she liked it:

> "Actually, I think [hierarchy is] useful. I'll use it to sort of organize, so that if I have to change, like, values for something. Like for the face. I did the eyes in one thing and the nose in one thing. So if I needed to change something, I could go try to find it. It worked."

Lois had very good strategies for beginning programs. In her protocol, for example, she was asked to draw a square and move it (using animation techniques) across the screen. The first thing she did after reading the task assignment was to write a program to draw the square. Though she was unsure how to do the animation, she knew how to perform that part of the program, so her heuristic was to do the part that she knew how to do, and to develop the program from there.

> "Oh, now there's a square . . . now I just have to move it? This is what I don't know how to do. . . . It would help if I could have my notes, my old worksheets.

Composition

Lois had no problems ordering and composing plans. She used well-named data objects and seemed to have little trouble in finding data objects.

Debugging

Soon after the midsemester interview, Lois became very interested in one of a the assigned graphics programs, "Triangles." During the first eight weeks of class, Lois ran her programs an average of thirteen times per day. While working on "Triangles" and the "Challenge," she ran her programs an average of thirty-two times per day.

Her class programs began improving significantly, according to the instructor. She completed programs as fast or faster than Allen. For example, on one particularly difficult graphics-oriented program, Allen took two days, Fred took two days, and Lois took one day.

Lois used good debugging strategies during her protocol. Whenever bugs occurred, she stopped writing her program and mentally simulated it, stating predictions for what should happen.

> "It didn't work! No problem—I can deal with it . . . I have the framerect first, and it should be going down. So the right should be 500."

GPCeditor Use

Lois did not use the library much during the first part of the semester. In fact, during the midsemester interview, she claimed that she did not know how, though her usage trace files indicated that she had saved and reused a plan. However, she began saving, reusing, and tailoring plans by the end of the semester. Lois made

little use of alternative representations. She relied mostly on the code view and the goal-plan buckets.

Sample Program

Lois's "Challenge" program and decomposition are in Exhibit C. Her program is well-composed with good data object naming. Her program has the same clear, repetitive structure of Sue's program but does not have the unnecessary redundancy that Sue had.

Her decomposition reflects the stage of her learning at the time of the "Challenge" program. She understood hierarchy and plan reuse but had not fully generalized it yet. Her first triangle (drawn horizontally) of the "Challenge" is created under the `Create the Loop`. This same plan is reused and tailored to draw the other horizontal triangle when achieving the goal `Do 3rd Triangle`. However, Lois did not seem to recognize that the `Create the Loop` plan could be tailored to draw vertical triangles as well. She used a flat hierarchy for drawing the second triangle under the plan `Init. Values2`, and then repeated the same flat order of plans to draw the fourth triangle (the other vertical one).

Fred

Student Characteristics

Fred entered the class as a freshman with extensive computer experience. He already had programmed in Basic, Pascal, and a graphics-oriented version of Pascal called Grasp. However, the longest program he had ever written previous to this class was only 20 lines long. He took the class because he wanted to learn more about Pascal.

Use of Reflection and Explicit Planning

Fred began the semester reflecting early and often. In his first protocol, Fred carefully considered what he should do in the program, stated constraints, and reflected on his program during debugging. When he found that he was forgetting what he wanted to do, he asked for pencil and paper to keep notes.

> "I'm thinking of doing a paintcircle for the face with two invertcircles for the eyes and mouth. No, I don't think I will."
>
> "The horizontal . . . the vertical should be the same as the second vertical because if they were different they'd be in the wrong place."
>
> (*Clicks with mouse on right eye of face he's drawn so far.*) "If that's 600 . . ." (*clicks on left eye*) "And that's 100, no, 300 . . ." (*Moves mouse arrow up and down where the nose is going to be drawn.*) "Then that's the horizontal. The vertical will be . . . 500. About 500,500 ought to do it."
>
> "Okay, I'll do a match to see what the numbers actually are . . . Aw, geez, I can't remember. Can I have a pencil so I can write some of this down?"

(Unfortunately, his strategy for using pencil-and-paper notes was incomplete. As he made changes, he forgot to update his notes. When he later modified his program based on his now inaccurate notes, he only implemented more bugs.)

Decomposition

Fred had little trouble starting programs, but he used none of the decomposition design techniques discussed in class. He rarely used plan groupings to create new levels of hierarchy and he never saved a plan to the library during the entire semester.

At one point in the first protocol, Fred created a new level of hierarchy. But when he found a bug, he deleted the entire level with several plans defined within it and then continued writing nonhierarchical code. He did not seem to understand or trust hierarchy.

Although he did not really take advantage of the structure within the GPCeditor, Fred seemed to appreciate it. During the midsemester interview, Fred said that he found goals and plans to be useful.

> "I think (the GPCeditor) is okay. The first couple of weeks, I thought it was kind of lame, that goals and plans got in the way. But I can see that it's kind of helpful. I can see that in longer programs, it's kind of helpful."

Composition

Fred had several problems with composition. During his first protocol, he frequently confused the composition operations. Fred had significant problems with data handling in both his first and second protocols. He referred to his data objects in his protocols by their values instead of by their meanings, especially during the first protocol.

> (*Pointing at the mouth of his face.*) "That's 500. It should be 450. 450 . . . 650. But which is which? That's 450 (*moving mouse up and down*) and that's 650 (*moving it side to side*). I'll put down 450 before I forget."

As the programs grew larger, Fred found it difficult to remember all the components and the meaning of terms like horizontal and vertical. While in the first program of the first protocol, he named his data objects meaningfully (e.g., `radius` and `horizontal`, but named his data objects in the second program of the first protocol based on their values (e.g., `sevenhundred` and `four-hundredfiftey`). This made it more difficult for him to remember the meaning of these data and to use them when instantiating plans for the program.

By the final protocol, he had learned to keep track of his variables, even with his poor naming schemes. Though he did not verbalize these strategies, he showed no difficulty in manipulating his data objects. In general, Fred's programs did work. In fact, he occasionally exceeded program requirements, adding interesting, new features.

Debugging

Fred had a great deal of difficulty with debugging. In part, his problem was due to forgetting data object meanings. In addition, however, he also forgot his goals in debugging. He would frequently begin work on a bug, switch his focus to a new bug, and forget about correcting the original bug.

GPCeditor Use

Fred never saved a plan to the library, rarely used any hierarchy, and found the overview useless. In general, he only used the code view and the program output when building or debugging his programs.

"No, [the overview] gets in the way."

Do you save things into the library?

"No, I don't remember how."

Sample Program

Fred's program and decomposition appear in Exhibit D. His program has a clear structure that shows the repetitive nature of the task. However, his data objects are so confused that one wonders how he was able to debug this program. The constant `fiftey` has the value of 200.0, the constant `one` has the value of 5.0, and his variable names are `z`, `y`, `x`, and `vari`.

His decomposition is almost perfectly flat. He uses a plan grouping once to link a `while-do` loop with its test expression. The rest of the program is not hierarchically decomposed and is difficult to understand.

EXHIBIT A: SUE'S CHALLENGE PROGRAM AND DECOMPOSITION

```
PROGRAM challenge;

     CONST

          one_hundred = 100.0;
          write = 'Enter the amount of spaces you would like between the
          lines.';
          two_hundred = 200.0;
          zero = 0.0;

     VAR

          center : REAL;
          read : REAL;
          vertical_start : REAL;
          Horizontal_start : REAL;
          vertical : REAL;
          horizontal : REAL;

BEGIN
     horizontal := zero;
     vertical := zero;
     Horizontal_start := two_hundred;
     vertical_start := two_hundred;
     WRITELN (write) ;
     READLN (read) ;
     WHILE (horizontal <= two_hundred) DO
          BEGIN
               center := one_hundred;
               MOVETO (horizontal, vertical);
               LINETO (one_hundred, one_hundred);
               horizontal := (horizontal + read);
          END;
     WHILE (vertical <= two_hundred) DO
          BEGIN
               center := one_hundred;
               MOVETO (horizontal, vertical);
               LINETO (one_hundred, one_hundred);
               vertical := (vertical + read);
          END
     WHILE (horizontal >= zero) DO
          BEGIN
               center := one_hundred;
               MOVETO (horizontal, vertical);
               LINETO (one_hundred, one_hundred);
               horizontal := (horizontal - read);
          END;
     WHILE (vertical >= zero) DO
          BEGIN
               center := one_hundred;
               MOVETO (horizontal, vertical);
               LINETO (one_hundred, one_hundred);
               vertical := (vertical - read);
          END.

END;
```

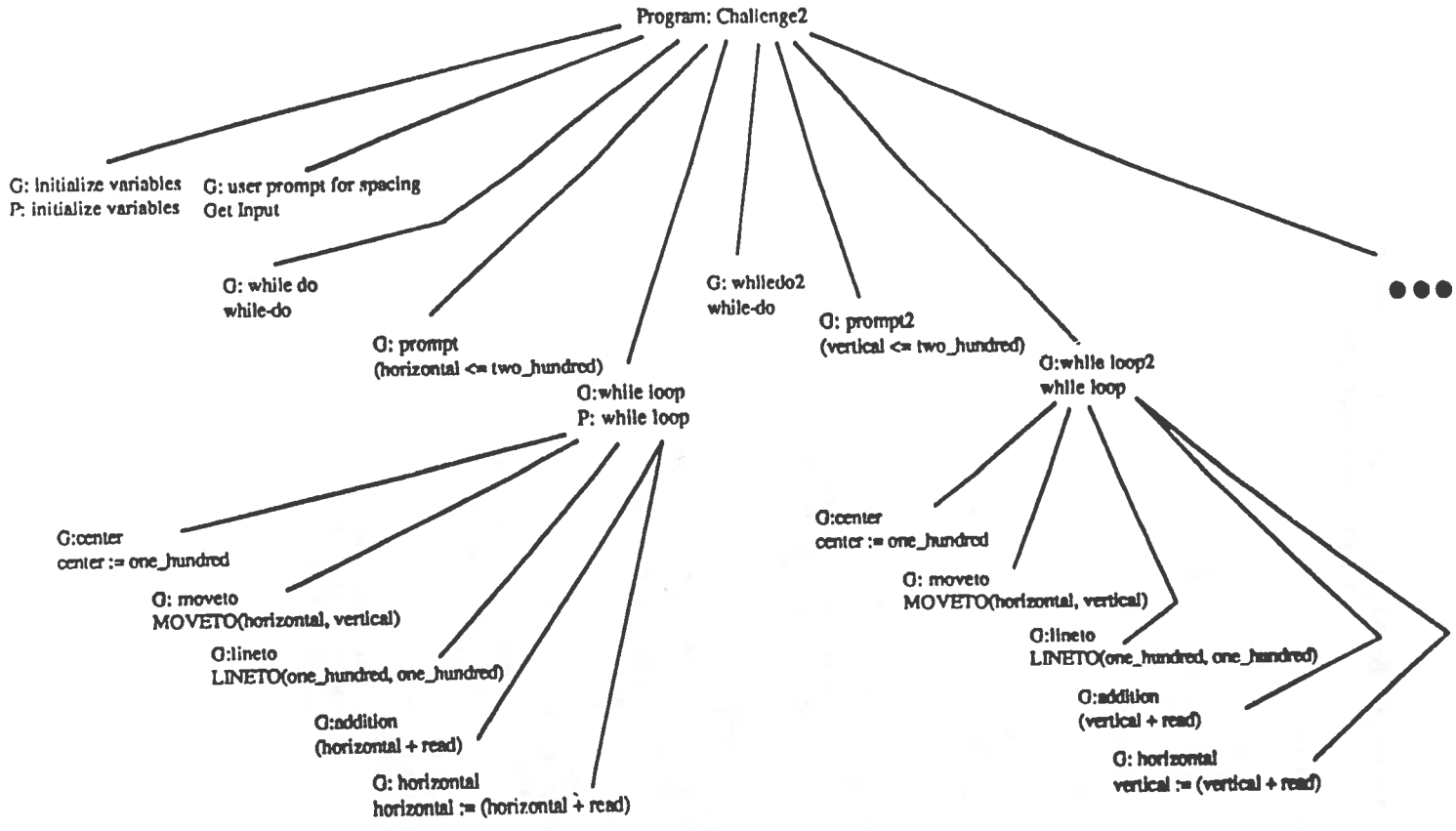

Program: Challenge2
G: Initialize variables
P: initialize variables
G: user prompt for spacing
Get Input
G: while do
while-do
G: prompt
(horizontal <= two_hundred)
G:while loop
P: while loop
G: whiledo2
while-do
G: prompt2
(vertical <= two_hundred)
G:while loop2
while loop
G:center
center := one_hundred
G: moveto
MOVETO(horizontal, vertical)
G:lineto
LINETO(one_hundred, one_hundred)
G:addition
(horizontal + read)
G: horizontal
horizontal := (horizontal + read)
G:center
center := one_hundred
G: moveto
MOVETO(horizontal, vertical)
G:lineto
LINETO(one_hundred, one_hundred)
G:addition
(vertical + read)
G: horizontal
vertical := (vertical + read)

EXHIBIT B: ALLEN'S CHALLENGE PROGRAM AND DECOMPOSITION

```
PROGRAM chalgenge;

     CONST
          cont_0 = 0.0;
          vert_400 = 400.0;
          vert_400_b = 400.0;
          five = 5.0;
          vert_linto = 0.0;
          vertcal = 200.0;
          hort = 200.0;
          con_400 = 400.0;

     VAR
          z : REAL;
          B : REAL;
          A : REAL;
          x : REAL;

BEGIN
     WHILE (x < con_400) DO
          BEGIN
               MOVETO (hort, vertcal);
               LINETO (x, vert_linto);
               x := (x + five);
          END
     A := con_400;
     B := con_400_b;
     WHILE (z < vert_400) DO
          BEGIN
               MOVETO (hort, vertcal);
               LINETO (con_400, z);
               z := (z + five);
          END;
     WHILE (A > cont_0) DO
          BEGIN
               MOVETO (hort, vertcal);
               LINETO (A, vert_linto);
               A := (A - five);
          END;
     WHILE (B > vert_linto) DO
          BEGIN
               MOVETO (hort, vertcal);
               LINETO (cont_0, B);
               B := (B - five);
          END;

END.
```

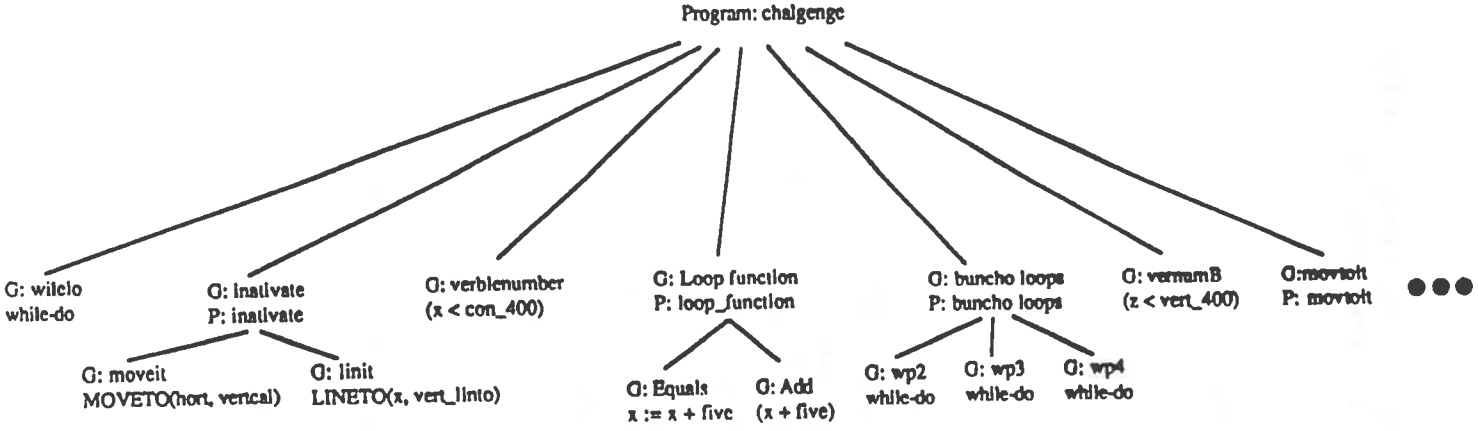

Program: chalgenge
G: wilelo
while-do
G: inativate
P: inativate
G: verblenumber
(x < con_400)
G: Loop function
P: loop_function
G: buncho loops
P: buncho loops
G: vermamB
(z < vert_400)
G:movtoit
P: movtoit
G: moveit
MOVETO(hort, vertcal)
G: linit
LINETO(x, vert_linto)
G: Equals
x := x + five
G: Add
(x + five)
G: wp2
while-do
G: wp3
while-do
G: wp4
while-do

EXHIBIT C: LOIS'S CHALLENGE PROGRAM AND DECOMPOSITION

```
PROGRAM Doing_the_Challenge;

     CONST
          five = 5.0;
          four_hundred = 400.0;
          two_hundred = 200.0;
          zero = 0.0;

     VAR
     final_vert : REAL;
     final_hori : REAL;
     vert : REAL;
     hori : REAL;

BEGIN
     hori := zero;
     vert := zero;
     final_hori := two_hundred;
     final_vert := two_hundred;
     WHILE (hori <= four_hundred) DO
          BEGIN
               MOVETO (hori, vert);
               LINETO (final_hori, final_vert);
               hori := (hori + five);
          END
     WHILE (vert <= four_hundred) DO
          BEGIN
               MOVETO (hori, vert);
               LINETO (final_hori, final_vert);
               vert := (vert + five);
          END
     WHILE (hori >= zero) DO
          BEGIN
               MOVETO (hori, vert);
               LINETO (final_hori, final_vert);
               hori := (hori + five);
          END
     WHILE (vert >= zero) DO
          BEGIN
               MOVETO (hori, vert);
               LINETO (final_hori, final_vert);
               vert := (vert - five);
          END;

END.
```

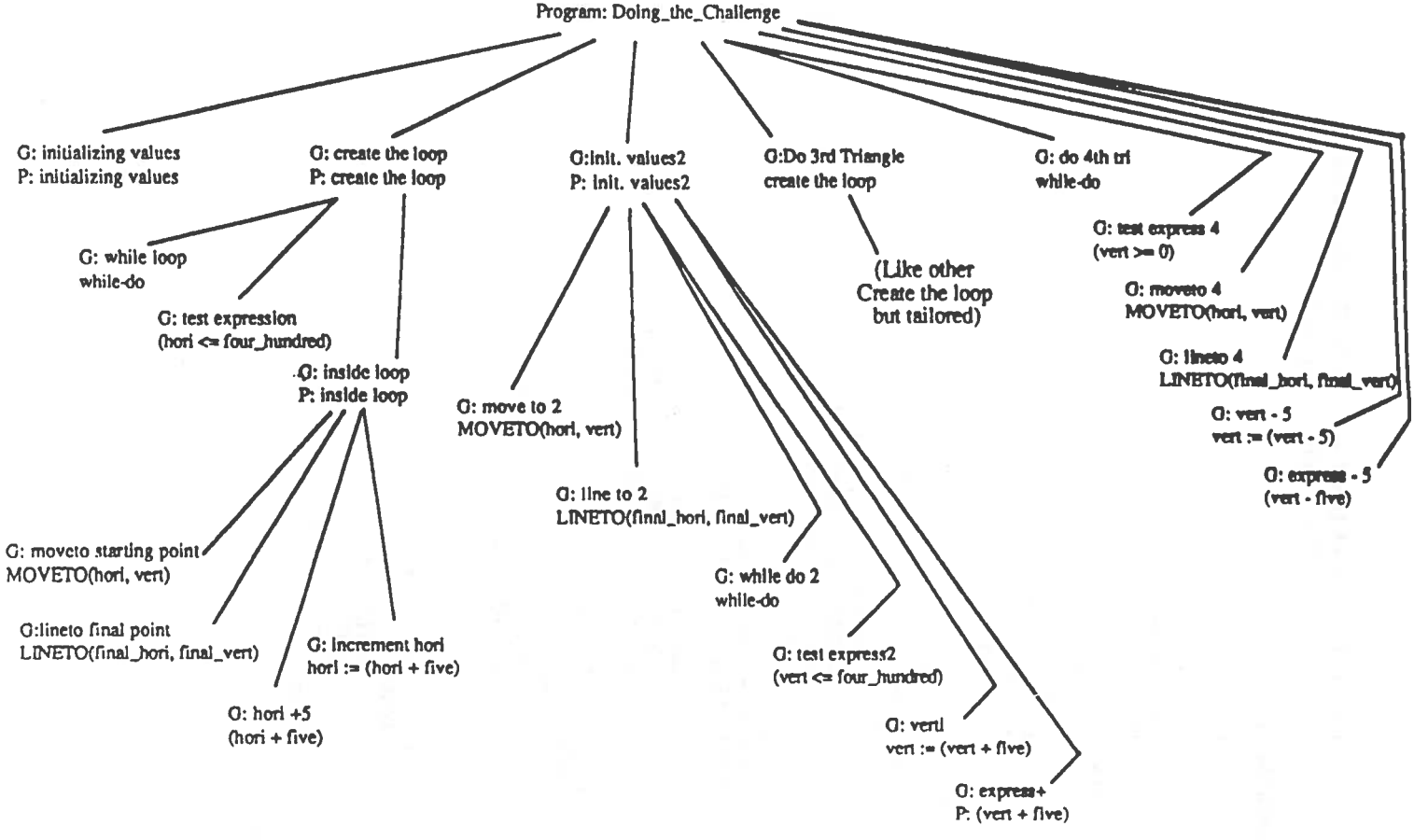

Program: Doing_the_Challenge
G: initializing values
P: initializing values
G: create the loop
P: create the loop
G:init. values2
P: init. values2
G:Do 3rd Triangle
create the loop
G: do 4th tri
while-do
G: while loop
while-do
G: test expression
(hori <= four_hundred)
G: inside loop
P: inside loop
(Like other
Create the loop
but tailored)
G: test express 4
(vert >= 0)
G: moveto 4
MOVETO(hori, vert)
G: lineto 4
LINETO(final_hori, final_vert)
G: vert - 5
vert := (vert - 5)
G: express - 5
(vert - five)
G: move to 2
MOVETO(hori, vert)
G: line to 2
LINETO(final_hori, final_vert)
G: while do 2
while-do
G: test express2
(vert <= four_hundred)
G: vert
vert := (vert + five)
G: express+
P: (vert + five)
G: moveto starting point
MOVETO(hori, vert)
G:lineto final point
LINETO(final_hori, final_vert)
G: hori +5
(hori + five)
G: increment hori
hori := (hori + five)

EXHIBIT D: FRED'S CHALLENGE PROGRAM AND DECOMPOSITION

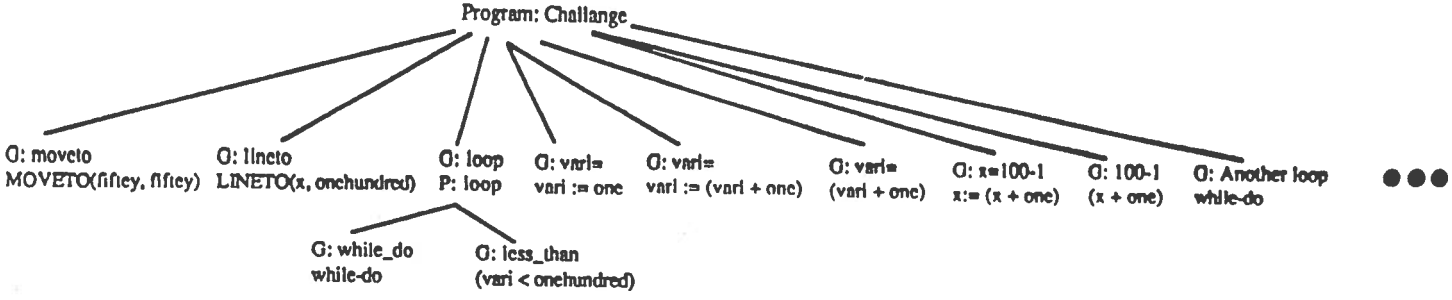
```
PROGRAM Challange;

     CONST
          zero = 0.0;
          fiftey = 200.0;
          onehundred = 400.0;
          one = 5.0;

     VAR
          z : REAL;
          y : REAL;
          plusten : REAL;
          z : REAL;
          vari : REAL;

BEGIN
     vari := one;
     WHILE (vari < onehundred) DO
          BEGIN
               MOVETO (fiftey, fiftey);
               LINETO (x, onehundred);
               vari := (vari + one);
               x := (x + one);
          END
     plusten := onehundred;
     WHILE (plusten >= zero) DO
          BEGIN
               MOVETO (fiftey, fiftey);
               LINETO (onehundred, plusten);
               plusten := (plusten - one);
          END
     y := onehundred;
     WHILE (y >= zero) DO
          BEGIN
               MOVETO (fiftey, fiftey);
               LINETO (y, zero);
               y := (y - one);
          END
     z := zero;
     WHILE (z <= onehundred) DO
          BEGIN
               MOVETO (fiftey, fiftey);
               LINETO (zero, z);
               z := (z + one);
          END;

END.
```


Program: Challange
O: moveto
MOVETO(fifiey, fifiey)
O: lineto
LINETO(x, onehundred)
O: loop
P: loop
O: vari=
vari := one
O: vari=
vari := (vari + one)
O: vari=
(vari + one)
O: x=100-1
x:= (x + one)
O: 100-1
(x + one)
O: Another loop
while-do
G: while_do
while-do
O: less_than
(vari < onehundred)

Feb 2

F23

N

LEARNING TO DESIGN, DESIGNING TO LEARN: Using Technology to Transform the Curriculum

Edited by
Diane P. Balestri
Princeton University

Stephen C. Ehrmann
The Annenberg/CPB Project

David L. Ferguson
State University of New York–Stony Brook

Taylor & Francis
Washington • Philadelphia • London