Supporting Pascal Programming with an On-line Template Library and Case Studies

Patricia K. Schank, Marcia C. Linn, and Michael J. Clancy

Education in Mathematics, Science and Technology
Graduate School of Education
University of California, Berkeley, CA 94720
schank@garnet.berkeley.edu, (510) 654-8931
mclinn@violet.berkeley.edu, (510) 643-6379

Electrical Engineering and Computer Sciences
Evans Hall
University of California, Berkeley, CA 94720
clancy@ucbarpa.berkeley.edu, (510) 642-7017

Abstract[1]

We propose a *template library* as a good representation of programming knowledge, and programming case studies as part of an effective context for illustrating design skills and strategies for utilizing this knowledge. In this project, we devised an on-line network of Pascal programming *templates* called a template library, and tested it with subjects (classified as novice, intermediate, and expert Pascal programmers) both as a stand alone resource and in conjunction with programming case studies. We investigated three questions using these tools: 1) How do subjects organize templates? 2) How well can subjects understand and locate templates in the template library? 3) Does the template library help subjects reuse templates to solve new problems? Results suggest that the template representations helped subjects remember and reuse information, and that subjects gained deeper understandings if the representation was introduced in the context of a programming case study.

---

[1]This material is based upon research conducted by the first author in partial satisfaction of the requirements for the degree of Master of Science in the Department of Electrical Engineering and Computer Sciences, University of California, Berkeley, May 1989, and is supported by the National Science Foundation under grant MDR-8954753. Any opinions, findings, and conclusions or recommendations expressed in this publication are those of the authors and do not necessarily reflect the views of the National Science Foundation.

Introduction

Experienced programmers' knowledge has been characterized as being *schematic* in that it consists of sequences of stereotypic actions (e.g., Soloway & Ehrlich, 1984; De´tienne & Soloway, 1990), *cohesive* in that it is interconnected in multiple ways, and *robust* in that it is stored abstractly along with strategies and conditions for applying the knowledge to solve new problems (e.g., Eylon & Helfman, in press; Linn & Clancy, 1992b). We have modeled these characterizations in what we call a *template* representation of Pascal programming knowledge that interconnects multiple representations of problem solutions (including schematic plans, verbal descriptions, pseudocode, Pascal code examples, illustrations, and animations) with debugging and testing strategies (see Figure 1). In this project, we created an on-line Pascal template library, and examined how novice, intermediate, and expert Pascal programmers organize, learn, and apply templates. Specifically, we ask three questions: 1) How do programmers organize templates? To assess this, we asked subjects in Study One to sort Pascal problem solutions by template. 2) Can programmers learn templates from a template library? To investigate this, we observed how subjects in Study Two used the library to understand and locate templates. 3) Does a template library help programmers apply and reuse templates to solve new problems? To assess this, we asked subjects in Study Three to solve programming problems using templates in the library, and observed which templates were used and how they were used.

The template library provides the building blocks for program design. To illustrate the process of design, we used Clancy and Linn's (1992) case studies of Pascal programming problems. These case studies introduce templates in context, illustrate design decisions for complex programming problems, demonstrate how templates are linked to design decisions, and discuss both effective and ineffective strategies for problem solution. We propose that using a database of programming templates, introduced in a case study context, will help students build effective representations of programming knowledge that they can apply to solve other complex problems.

—Insert Figure 1 about here —

Rationale

The template library includes information generally taught in introductory Pascal courses and texts (e.g., Cooper & Clancy, 1985; Dale and Orshalick, 1983; Clancy & Linn, 1992). We built the library and refined its content and organization at length over a year of weekly research meetings and as a result of pilot work (see Schank, 1989).

WHY TEMPLATES?

Research in several domains such as chess (e.g., Anderson, 1986), physics (e.g., Larkin, McDermott, Simon, & Simon, 1980), and programming (e.g., Adelson & Soloway, 1985; Soloway, 1985) suggests that experts organize their knowledge in larger, cohesive schematic structures than novices, who tend to possess a more syntactic representation of programming knowledge. Applying these findings to programming, several studies suggest that representing programming knowledge in the form of short programming plans also helps students write programs (e.g., Anderson & Reiser, 1985; Linn & Dalbey, 1985). Effective knowledge representations are also often characterized as (a) emphasizing salient features, (b) being abstract enough to facilitate analogies and problem solving within and across domains, (c) cohesively integrating information in multiple ways, and (d) being robust in that information is stored abstractly along with strategies and conditions for applying the information to solve new problems (e.g., Eylon & Helfman, in press; Linn & Clancy, 1992a, 1992b). We designed our Pascal templates to capture these characterizations, in effort to help students build effective knowledge representations that would help them solve other problems. To communicate the template representation to students and to help them internalize it, our library (a) includes multiple representations of programming information, (b) illustrates how this information is related, and (c) provides students with ways to easily enter new information and modify existing information. We asked subjects in Study One to sort templates to help us assess our template organization scheme.

WHY A HYPERMEDIA TEMPLATE LIBRARY?

For the template model to help students, it must be presented clearly and in a way that simplifies the process of understanding (e.g., presentation should minimize cognitive load while maintaining students' foci; Baecker & Buxton, 1987). We used several tactics to minimize interference, facilitate chunking, and reduce cognitive load. For example, we used a library metaphor to help students understand the model, since libraries are familiar and people typically know how to use them. Since we wanted to offer multiple template representations (e.g., verbal descriptions, pseudocode, code, illustrations, animations) and interconnections between templates, we chose a hypermedia[2] representation for the library. Some learners may feel overwhelmed by the complexity of hypermedia documents and prefer more structured (e.g., linear) environments (e.g., Smith & Weiss, 1988). Others, however, benefit from, and do better in, exploratory environments that let them manage their learning (e.g., Recker & Pirolli, 1992). Neither type of environment is inherently better; rather, each has its merits. Thus we designed the template library to support both structured and

[2]Hypermedia frameworks support the integration of textual, graphical, auditory, and video information in a network of linked nodes that allow multiple paths and cross-references when they are relevant.

exploratory learning strategies via multiple on–line browsing tools. For example, Clancy and Linn's (1992) case studies provide a structured, relevant context for learning the templates in the library (See Appendix A for a sample case study problem statement.) The template catalogs and keyword search mechanisms support more exploratory strategies. In Study Two, we examine the roles of (a) multiple representations and links for understanding templates, and (b) browsing tools (e.g., case studies, template catalogs, search mechanisms) for locating templates.

WHY CASE STUDIES?

Clancy and Linn's (1992) Pascal case studies explicate strategies and procedures for solving relatively complex problems. For example, they identify complex templates, specify how the templates can be applied and tested, identify relations between templates, explore alternate approaches for solving a problem, and encourage reflection. Empirical tests of these case studies provided evidence that the expert commentary helped students learn program design skills (Linn & Clancy, 1992a, 1992b). Templates introduced in these case studies suggest ways to decompose problems (e.g., into their component templates), helping to reduce the cognitive demands of programming (Linn & Dalbey, 1985) so students can write more complicated programs. We hypothesized that giving students a structured, cohesive template model would encourage them to generate self-explanations and help them solve programming problems. We presumed that presenting the template library in conjunction with design strategies illustrated in the case studies would encourage them to build robust and cohesive representations of programming information that they could apply to solve new problems. In Study Three, we examine how subjects familiar with case studies use templates to solve programming problems.

Implementation of the Pascal template library

The template library contains approximately seventy templates of algorithms generally taught in introductory Pascal programming courses, including templates introduced in the Block Letters, Calendar, Roman Numerals, and Text Justification Pascal case studies (Clancy & Linn, 1992) and links from case study decomposition diagrams to these templates. (See Table 1 for a list of templates in the library.) We implemented the library in Hypercard because it provides support for various media (e.g., text, video, and graphics) and the creation of cross–reference links within and between media, as well as tools to facilitate rapid and easy interface prototyping.

—Insert Table 1 about here—

INTERFACE DESIGN

We use a public library interface metaphor so students can apply familiar strategies for searching and categorizing library materials to the template library (see Figure 2), and provide a library map to help students navigate through the library (see Figure 3). Students can see different representations of a template (e.g., verbal descriptions, pseudocode, code, and illustrations; animations are also available for about 20 of the templates) by clicking on the appropriate icons (see Figure 4). Since students may prefer certain representations over others, they can specify which representation they would like to see first when a template is selected. (The standard default representation is pseudocode.)

—Insert Figures 2-4 about here—

LIBRARY FEATURES

Students can obtain on-line help, take a quiz, or examine templates in the library. The templates are multiply indexed by data structure, control flow, program design, or case study. Templates can also be accessed by name using a provided keyword search mechanism.

On–line help. On-line help is available from any point in the library. To obtain help, students can click on the librarian (in the front lobby), or the librarian icon (located on the lower left corner of every screen). Information is available on several topics, including: library registration, general navigation techniques, the library layout, template indexing schemes, case studies, search mechanisms, and the template quiz. To go to any section of the library, students can simply select the desired section from the library map (accessible from all screens).

Case studies. To access case study templates, students can click on the bookshelf in the front lobby of the library and select the book with the appropriate case study name displayed on its binding. A decomposition diagram of the case study programming problem is then displayed (see Figure 5). Students can click on any part of the diagram to see the template(s) used at a particular stage of the problem decomposition.

—Insert Figure 5 about here—

Template catalogs and combinations. Templates in the library are indexed in several ways: by data structure (e.g., single variables, arrays, files), control flow (e.g., sequential, conditional, looping controls), and general global program design (e.g., "bulk" processing—serially reading and storing all data in structures such as arrays before processing them—or "one-by-one" processing—iteratively reading and processing items one at at time without use of structures such as arrays). To access templates, students can select the template catalog and choose the program design

drawer, data structure drawer, or control flow drawer (see Figure 6). To search for templates that employ particular combinations of decomposition, data structure, and control flow, (e.g., to view a template that combines bulk processing of data with use of arrays and loops), students can click the Combo button, and select the features they are interested in. A list of relevant templates is then displayed, a template can be selected from this list.

—Insert Figure 6 about here—

Direct template access and keyword search. Students may want to circumvent the template catalog and access templates directly if they are familiar with the library or are interested only in a specific template (e.g., such as Bubble Sort). To do this, students can click on the computer terminal in the front lobby and then (a) select a particular template from a list of all templates in the library, or (b) enter the the partial (or entire) title name of a template to initiate a search..

Template quiz. To test their general knowledge of templates in the library, students can click on the template quiz notebook displayed on the front desk of the library lobby. Alternately, when viewing a template, students can test themselves on that particular template by selecting the Quiz Question option under the Template Info menu. Quiz questions prompt students to choose one of two Pascal code segments as the better example of the current template, and students can request a "hint" whereupon the pseudocode representation of the template is displayed. Upon finishing, the number of requested hints, correct answers, and incorrect answers are reported.

TEMPLATE REPRESENTATIONS AND RELATED TEMPLATE LINKS

Several representations of the template action are available in each programming template (see Figure 1), entitled Pseudocode, Verbal Description, Sub-Pseudocode, Pascal Code, Illustrations (both static and dynamic) and Bugs & Testing in the library (see Figures 7–8). To encourage elaborations, templates related to the current template can be accessed via the List Template Links menu. For example, students can compare the pseudocode of a related template side-by-side with the pseudocode of the current template, and/or go to the related template. Related templates can be chosen from the list of All Templates, More Specific Templates, More General Templates, Elaborated Templates, or Super-Templates. More general templates are templates with more abstract pseudocode and function, more specific templates are templates with more detailed pseudocode and function, elaborated templates are templates using specific data or control structures, and super-templates are templates that "contain" the current template. For example, consider a template for inserting an item into an array: A more general template inserts an item into any arbitrary sequence, a more specific template inserts an item in increasing order, an elaborated template inserts an item into a linked list, and a super-template is an insertion sort (see Figure 1).

—Insert Figures 7-8 about here—

To encourage students to reflect on their understanding of Pascal programming, we provide support for modifying (i.e., personalizing) the library. For example, virtually every portion of the database (e.g., the template information, and links between templates) can be modified via the Edit menu. Via the Template Info menu, students can create, delete, and save templates, set the default template representation, take a quiz question, and find out (or specify) what data structure(s), control flow(s), or program design scheme each template employs.

Method

Ten volunteers from the University of California, Berkeley participated in the study, including six novice, two intermediate, and two expert programmers. The novice programmers (defined as students currently enrolled in a college freshman level introductory Pascal course) included two women and four men. The intermediate programmers (defined as students with 2-3 semesters of programming experience, at least one of which was in Pascal) included one male senior and one female second year graduate student. The expert programmers (defined as students with 3 or more semesters of programming experience) included two male engineering graduate students. Novices were introduced to the library after having read paper versions of the case studies. Intermediates and experts were unfamiliar with the case studies. Each subject met individually with the experimenter for 2 to 8 hours to solve problems involving the templates while "talking aloud." (Students were introduced to the library during the first 30 minutes of the first session. See Table 2 for sample instructions and problems given in Studies One, Two, and Three.)

—Insert Tables 2 and 3 about here—

Study One: How do programmers organize templates?

As mentioned, research suggests that novice programmers often have more surface level, or syntactic, representations of programming knowledge, while experts tend to organize their programming knowledge in more abstract conceptual structures. We represent these structures as templates. Programmers often have templates for sorting, accumulating, pattern matching, and many other actions. What templates do experts, intermediates, and novice programmers see as being in the same categories? Do the experts sort in a way consistent with the organization of the template library (e.g., by data structure, control flow, or global program design)? How do the novices sort the templates, given their limited familiarity with them? Answers to these questions provide guidance for possible reorganization of the library. To address these questions, we asked two novice, two

intermediate, and two expert programmers to sort a stack of 15-20 templates (see Table 2, Study One).

STUDY ONE RESULTS AND DISCUSSION

We found that intermediates and experts sorted the templates into fewer, and more abstract, categories, and converged quickly to a final set of categories after examining only a few templates. Novices sorted templates into more syntactical (surface-feature) categories, and had more "leftover" unlabeled categories (i.e., categories with templates whose purposes were not clear to them; see Table 3, Study Two Results). For example, one novice invented eight categories, three of which were unlabeled, and the other novice had five categories, with two unlabeled. The categorical labels that novices invented were more likely to highlight syntactical surface features (e.g., "array" and "case"). The intermediates used fewer categories, and tended to use more abstract, and fewer syntactical, categorical labels than did the novices. The experts used the fewest categories, had no unlabeled categories, and used the most abstract categorical names (e.g., "loop until condition").

Assuming that the subjects sorted the templates into a minimal number of categories, the observation that the intermediates and experts sorted templates into fewer, more abstract categories than novices supports the notion that they possess a higher level organization that enables them to more easily see deep similarities between the templates. For instance, one expert sorted the templates by program design strategy ("independent data" vs. "dependent data"), while the other sorted by the (somewhat orthogonal) category of control flow ("fixed loop" vs. "loop until condition" vs. "sequential"), both category choices that were similar to those implemented in the library by the expert designers (see Figure 6). Subjects sorted the templates into several different categories, however, suggesting that is is beneficial to provide on–line support to students for modifying (e.g., reorganizing) the library as they learn templates.

Study Two: Can subjects learn templates from the library?

In the second study, we ask two main questions: 1) How do subjects learn new templates? Will the multiple representations and related templates help, and how will they be used? 2) How do subjects search for templates in the library? In particular, how do novice programmers (who read paper versions of the case studies) use the library, compared to the experts and intermediates (who were unfamiliar with the case studies)? How do subjects differentially use the indexing methods of data structure, control flow, or program design strategy? To address these questions, all 10 subjects were shown a new template and instructed to use any of the available methods (i.e., such as looking at multiple representations and related templates) to help them understand the template, and were asked to find templates in the library (see Table 2, Study Two).

STUDY TWO RESULTS AND DISCUSSION

When introduced to a new template, subjects chose to view several representations. Subjects typically began by examining the pseudocode, found it too abstract, examined the Pascal example, found it too idiosyncratic, and then examined the verbal description or the illustration, and typically ended by returning to the pseudocode representation. Once a template was understood, subjects preferred to refer to it by the pseudocode. When asked to rate the template information in terms of its helpfulness for understanding the template action, the subjects overwhelmingly rated the pseudocode as the most helpful, although they found the alternate representations helpful also (see Table 3, Study Two Results). (The subjects didn't seem to find the sub-template pseudocode as helpful as we had hoped. For instance, one subject commented that it was "too difficult for me to divide my thoughts up.") Only two subjects, both of whom were novices, used related templates to help them understand the template at hand, and did so only after examining the pseudocode and verbal description. All but one subject preferred to use the pseudocode as the default representation. (One novice chose a default verbal representation.) Several subjects commented on how the library helped them make connections between different ways of thinking about the same problem solution, and indicated that juxtaposing multiple perspectives of a template helped them understand its purpose. For instance, one subject commented: "I really like the idea of being able to have a verbal description, and illustration, and pseudocode kind of all together cause you can look at it and go...yeah, I can kind of understand the pseudocode but then if you go back to the image you go 'ah! that's what that is.'"

When asked to find templates, novices usually attempted to find them via case studies. The novices generally found their templates more directly (and generally more quickly) than the intermediates and experts. Intermediates and experts preferred to search via the template catalog by data structure, control flow, or a combination of these features (see Table 3, Study Two Results). Few subjects used the computer search and even fewer took advantage of the program design index. Intermediates and experts searched for templates by data structures or control flow, rather than by the (more abstract) program design strategy, suggesting that experts are able to bring specific templates to mind when solving a problem.

In sum, subjects indicated that the multiple perspectives helped them understand the template solutions, and had some difficulty finding templates even if they were organized in predictable ways; even experts had some trouble using the organization developed by another expert. The library seemed more useful to those who had learned the templates in the context of case studies, suggesting that it is more useful to teach the templates in this context. We found it somewhat puzzling that, compared to novices, experts took about twice as many mouse clicks to find their templates. Possible reasons for this are: (a) case studies constrained the search space (and novices generally searched via case studies, and thus had fewer overall choices to make), (b) the solution

could be easily generated by experts so they were not motivated to locate templates, (c) the experts simply had so many possible template solutions in mind that they were searching for a specific optimal template, not just the first one that would do the job, or (d) inquisitive expert nature simply led experts to search down several paths. How experts might use the library to locate more complex templates (i.e., ones they could not generate so easily themselves) is an open question.

Study Three: Do subjects reuse templates to solve problems?

Does the template library help subjects reuse problem solutions? To answer this question, we asked four novices to solve a new problem, isomorphic to one of the case study problems, that could be solved using familiar templates from the library. The novices first learned about these templates while using the library in the context of the case studies. Two novices solved two problems each, and the other two solved one problem each. Subjects were instructed to use the library, if they wished, to solve each problem (see Table 2, Study Three).

STUDY THREE RESULTS AND DISCUSSION

Three of the four novices solved the problems by searching, via the case studies decomposition diagram, for a familiar template in the library, and one student generated the solution to a problem from scratch. The subjects who used the library searched for helpful templates via the case studies decomposition diagrams (see Table 3, Study Three Results). After finding a relevant template, all subjects looked at the code example that accompanied the template. Two of the subjects examined the pseudocode, matched the code example and pseudocode to understand the variables used in the example, and then substituted variable names appropriate for the new problem. Subjects often needed to a few new statements to the familiar template to solve the problem. All subjects added statements before or after the template, although more elegant solutions (e.g., solutions with fewer lines of code) involved adding statements within the template, suggesting that they conceptualized the templates as indivisible "chunks." In sum, subjects successfully reused the library templates that were introduced in the case studies. The solution generated by the one student who didn't use the library was isomorphic to the solutions by the subjects who did, suggesting that he had already internalized the solution represented by the template.

Summary

Template sorting results supported the hypothesis that experts represent programming knowledge in more abstract organizations than novices. In addition, the experts tended to group templates into categories similar to those implemented by the library designers. All subjects sorted the templates into different categories, suggesting that it is beneficial to provide on–line support to students for modifying and reorganizing the templates in the library. Several subjects commented on

how the library helped them make connections between different ways of thinking about the same problem, and used many of the available representations to help them understand a particular problem solution. Case studies provided a context for learning the library's template network model, helped novices re-use familiar templates in the library, and illustrated ways to decompose problems, helping novices solve more complicated problems without inventing new code. Subjects usually drew comparisons between the code and pseudocode representations to help them write the Pascal code to solve new problems. The library metaphor and the hypermedia environment, supplemented with the browsing tools such as the library map, template catalogs, keyword search, and case studies, helped support students' navigation through the library.

Further work

Programmers often face the task of learning new languages. While learning a second programming language is difficult, it is often easier than learning a first language because many schematic concepts and constructs are shared (e.g., Yu & Robertson, 1988, describe some of the shared constructs of Pascal and FORTRAN plan structures). Wiedenbeck and Scholtz (1990) found that skilled programmers learning a new language experience the most difficulty in the transition from tactical planning (language–independent local plans for solving problems) to implementation planning (language–dependent plans for implementing tactical plans in a particular language; Soloway, Ehrlich, Bonar, & Greenspan, 1984), and programmers' schematic representations form a base for the tactical/implementation planning cycle in which they access and revise their own schemas. Although we did not examine the transfer of programming skill between languages in this study, access to a large repertoire of templates may also facilitate transfer within similar types of languages (e.g., procedural, functional, object-oriented) because, as shared constructs, they form a basis for the interplay between tactical and implementation planning.

Based on the results of our studies with the Pascal template library, the Hypermedia Case Studies in Computer Science (HCS^2) Project developed a more flexible and extended database of programming examples (called a "perspective library") for the functional programming language Lisp, and used the library in conjunction with Lisp case studies (Linn, Katz, Clancy, & Recker, 1990) in an introductory Lisp course at the University of California, Berkeley. The Lisp perspective library, which represents information such as student notes, debugging information, and verbal descriptions as instances of *perspectives*, allows students to more easily reorganize the library to fit their own viewpoint. With the perspective library, students can create new perspectives and code examples, and include or exclude a particular code example in a given perspective. The HCS^2 group is currently developing an application to help students conveniently create, display, or hide particular comments in their Lisp programs, based upon categories like those implemented in the Pascal template and Lisp perspective libraries.

Acknowledgements

We especially thank Michael Ranney, Christopher Hoadley, Sherry Hsi, Marcia Lovett, the Reasoning Group, the HCS^2 Group, and two anonymous reviewers for their helpful suggestions and comments on earlier drafts.

References

Adelson, B. & Soloway, E. (1985). The role of domain experience in software design. *IEEE Transactions on Software Engineering, SE-11*(11), 1351-1360.

Anderson, J. & Reiser, B. (1985, April). The Lisp tutor. *Byte, 10*(4), 159-175.

Anderson, R.C. (1986). Some reflections on the acquisition of knowledge. *Educational Researcher, 13*(5), 5-10.

Baecker, R., & Buxton, W. (1987). *Readings in human-computer interaction: A multidisciplinary approach*. Los Altos, CA: Morgan Kaufmann Publishers, Inc.

Clancy, M. J., & Linn, M. C. (1992). *Designing Pascal solutions: A case study approach*. New York, NY: W. H. Freeman.

Cooper, D. & Clancy, M. (1985). *Oh! Pascal*. New York: W.W. Norton & Company, Inc.

Dale, N. & Orshalick, D. (1983). *Introduction to PASCAL and Structured Design*. New York: D. C. Heath and Co.

De´tienne, F. & Soloway, E. (1990). An empirically–derived control structure for the process of program understanding. *International Journal of Man–Machine Studies, 33*(3), 323-342.

Eylon, B. & Helfman, J. (in press). The role of examples, generalized procedures, and ability in solving physics problems. *Cognition and Instruction*.

Larkin, J.H., McDermott, J., Simon, D.P., & Simon, H. A. (1980). Expert and novice performance in solving physics problems. *Science, 208*, 1335-1342.

Linn, M & Clancy, M. (1992a).The case for case studies of programming problems. *Communications of the ACM, 35* , 3, 121-132.

Linn, M.C. & Clancy, M. (1992b). Can experts' explanations help students develop program design skills? *International Journal of Man-Machine Studies, 36,* 4, 511-551.

Linn, M. & Dalbey, J. (1985). Cognitive consequences of programming instruction: Access, and ability. *Educational Psychologist, 20* (40), 191-206.

Linn, M.C., Katz, M., Clancy, M. J., & Recker, M. (in press). How do Lisp programmers draw on previous experience to solve novel problems? In E. deCorte (Ed.). Computer Based Learning Environments, [Presented at NATO Conference Oct., 1990] Belgium: Springer Verlag.

Recker, M. & Pirolli, P. (1992). Student strategies for learning programming from a computational environment. Submitted to the *Second International Conference on Intelligent Tutoring Systems*.

Schank, P. (1989). *A Pascal template library*. Master of Science Research Project, Plan II. Department of Electrical Engineering and Computer Science, University of California, Berkeley.

Smith, J. B., & Weiss, S. F. (1988). An overview of hypertext. *Communications of the ACM, 31*(7), 816-819.

Soloway, E. (1985). From problems to programs via plans: The content and structure of knowledge for introductory Lisp programming. *Journal of Educational Computing Research. 1*(2), 157-172.

Soloway, E. & Ehrlich, K. (1984, Sept.). Empirical studies of programming knowledge. *IEEE Transactions on Software Engineering, 10*(5), 595-609.

Soloway, E., Ehrlich, K., Bonar, J. & Greenspan, J. (1984). What do novices know about programming? In A. Badre & B. Shneiderman, (Eds.). *Directions in Human–Computer Interaction*. Norwood, NJ: Ablex, 27-54.

Wiedenbeck, S. & Scholtz, J. (1990). Learning second and subsequent programming languages: A problem of transfer. *International Journal of Human-Computer Interaction, 2*(1), 51-72.

Yu, C. & Robertson, S. P. (1988). Plan-based representations of Pascal and FORTRAN code. In E. Soloway, D. Frye, & S. Sheppard [Special issue]. *CHI '88 Conference Proceedings: Human Factors in Computing Systems*.

Appendix A: Sample Case Study Problem Statement

Problem
Write a Pascal program that reads letters from the user and prints them down the screen in a block format. The program will first ask the user how large the letters will be, and read a number—the *letter size*—that indicates how many lines and columns should be used to print each letter. It will then ask the user for six letters chosen from P, A, S, C, and L. Finally, it will read and print the six letters, with printing done in a block format. A sample interaction (with the user input in boldface), and one of the block letters that the program might produce, are shown below.

```
How large should the letters be?  9
Type six letters, chosen from PASCL:  LACSAP

(block versions of L, A, C, S, A, and P follow)
```

```
*********
*********
**       **
**       **
*********
*********
**       **
**       **
**       **
```

Your program must include calls to one or more of four procedures: DrawBar, which prints asterisks completely across a specified number of lines; DrawLeft, which prints asterisks on the left side of a specified number of lines; DrawRight, which prints asterisks on the right side of a specified number of lines; and DrawLandR, which prints asterisks followed by blanks followed by asterisks, on a specified number of lines (like drawing a "left" and a "right" on those lines).

Tables and Figures

Table 1. Templates in the Pascal Template Library.

1. Bulk input/process/output	36. Left-right justify with arrays
2. One by one input/process/output	37. Print indent, left-right justify
3. Read & store values in array	38. Read item
4. Read & store all values	39. Process item
5. Prompt, read values into array	40. Draw a letter of the alphabet
6. Read, check, store in array	41. Draw the letter 'A'
7. Read file & store in array	42. Process item a fixed # of times
8. Read word & store in array	43. Process item: nested loops
9. Skip blanks in input	44. Keep loop counter
10. Read line & store in array	45. Print item
11. Read & store line	46. Print prompt/explanation
12. Initialize array	47. Print item with formatting
13. Frequency count using arrays	48. Print item a fixed # of times
14. Convert char digit to number	49. Print fixed number of blanks
15. Process array until condition	50. Print blank lines
16. Test if character is a letter	51. Read sequence of items
17. Convert letter to upper case	52. Prompt, read sequence
18. Process each array item	53. Prompt, read, error-check sequence
19. Find first array item w/property	54. Read sequence from file
20. Line translation using arrays	55. Process sequence
21. Accumulate array values	56. Process sequence until condition
22. Find average of array items	57. Frequency count sequence
23. Insert in array, increasing order	58. Deal with predecessor in sequence
24. Insert in array, maintaining order	59. Pick one of several actions
25. Process each 2D array item	60. Command interpreter
26. Bubble sort	61. Accumulate sequence
27. Insertion sort using arrays	62. Find average of sequence
28. Selection sort using arrays	63. Process a line of text
29. Selection sort	64. Process several lines of text
30. Switch two values	65. Basic text processing
31. Merge sorted sequences	66. Print sequence
32. Deal w/adjacent pairs in array	67. Left/right justify sequence
33. Print array values	68. Indent, left/right justify sequence
34. Print explanation & array	69. Print sequence to file
35. Print array values to file	

Table 2. Sample instructions and materials. (Comments in *italics* not seen by subjects.)

Study One verbal instructions:

- (Subjects given 15-20 templates, on paper). "Sort the given templates into categories that are meaningful to you -- as many categories as you like -- and state a) why you chose the categories that you did and b) how you decided to put what templates in what categories."

Study Two sample questions:

- (Shown the Print Indent, Left/Right Justify template): "Use any of the available methods to understand how this template works and what it does." *(Isomorphic to Calender).*

- Find templates in the library you could use to solve this problem:
  Read in a list of grades and count how many scores are in each of the following ranges: 0-9, 10-19, ... 80-89, 90-99, 100-109 (extra credit is possible!). The program should read in the numbers, calculate the frequency of scores in each range so that you could, for example, print a histogram. (*Subjects shown picture of a histogram.)*

- Find templates in the library you could use to solve this problem:
  Simulate a calculator program that asks the user to enter a command and them performs that task. For example, if the user enters the command "Add", the program asks for numbers and prints their sum. The user can enter commands like Add, Subtract, Multiply, Divide, Sqrt, and Quit. Assume that the procedures that ask for and perform these commands already exist; just find the template you would use to write the main program. *(Isomorphic to Block Letters.)*

Study Three sample problem statements:

- The Calendar case study uses the following pseudocode as its top level template:

  ```
  Get the input value {a single year}
  Process the input value {print the calendar for the year}
  ```

  Suppose we want to let the user to print several calendars without having to re-run the program (e.g., for years 1982, 1987, 1965). Assume that when the user enters a 0 the program should stop. Write the pseudocode to do this.

- Write the code for a procedure that accepts an indentation integer indent, a character ch, an integer n, and an integer width, and prints ch n times in a newspaper column format width characters wide, indented indent spaces on the first line. E.g.,*(Isomorphic to Text Justification.)*

  ```
    xxxxxxx
  xxxxxxxxx
  xxxxxxxxx
  xxxxxx....
  ```

Table 3. Results of Studies One, Two, and Three.

Study One, sorting results:

Group	Mean no. of categories	Mean no. unlabeled	Example names
Novices (2)	6.5	2.5	"switch values", "case", "array", "counter"
Intermediates (2)	5.5	0.5	"process", "control", "i/o", "loop", "array"
Experts (2)	2.5	0.0	"independent data", "dependent data", "loop until condition"

Study Two, searching and representations:

Group	Mean no. of clicks to find	Searched by	Min no. clicks required to find
Novices (6)	8	case studies	3
Intermediates (2)	11	catalog (data/control structure, or combo)	3
Experts (2)	16	catalog (data/control structure , or combo)	3

• Subjects representation preferences (in order of preference): 1) pseudocode, 2) code example & illustration, 3) bugs & testing, 4) students pseudocode & comments, 5) sub-template pseudocode.
• Typical progression: pseudocode -> example -> verbal description or illustration -> pseudocode.

Study Three, solving new problems:

Novices (4) typically solved a new problem by:
• Finding a template via the case study diagram in the template library,
• Comparing the templates' pseudocode and code example, and substituting variable names,
• Adding new lines above or below the template.

More general template pseudocode

Find the position for the new item
Make room for it by shifting other items
Put the new item in

Sub-template pseudocode

Shift array elements:
for i = last array index downto position + 1
arrayitem[i] <- arrayitem[i-1]

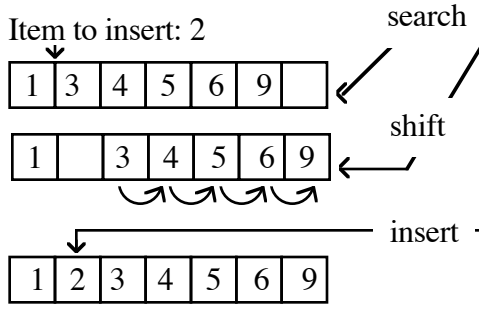
Super-template

Insertion sort:
For i = 2 downto number of items
begin
.....

Pseudocode

Find the position in the array for the item
For i = last array index downto position + 1
arrayitem[i] <- arrayitem[i-1]
array[position]=new item

Illustration



Pascal code example

```
type numbers=array[1..100] of real;
procedure Insert(var nums : numbers;
   var size : integer; newvalue : real);
var insertionSpot : integer;
begin
insertionSpot :=
    findbigger(newvalue, nums, size);
for i := size downto insertionSpot + 1
   do nums[i] := nums[i-1];
nums[insertionSpot] := newvalue;
size := size + 1;
end;
```

Verbal Description

Insertion into an array in increasing order is analogous to adding a new student to a classroom where students sit in alphabetical order. To add the new student, you could first determine where the new student should sit, and then ask the students from that seat on to shift down one seat.

Similarly, to insert an item into an array, first find out where the item belongs by scanning the array until you find the first item bigger than the new item. Then shift all the items in the array down one cell from that point on, and put the new item in the insertion spot.

Debugging and testing

Did you find the correct insertion spot, or are you off possibly by one? What if the new item is either bigger or smaller than all of the items in the array? What if the array is too small to hold the new item? Test using an array with:

- one, many, or no items initially
- all items bigger than the new item
- all items smaller than the new item

Figure 1. The Insert into Array in Increasing Order template, and some related templates (indicated by ---> lines).

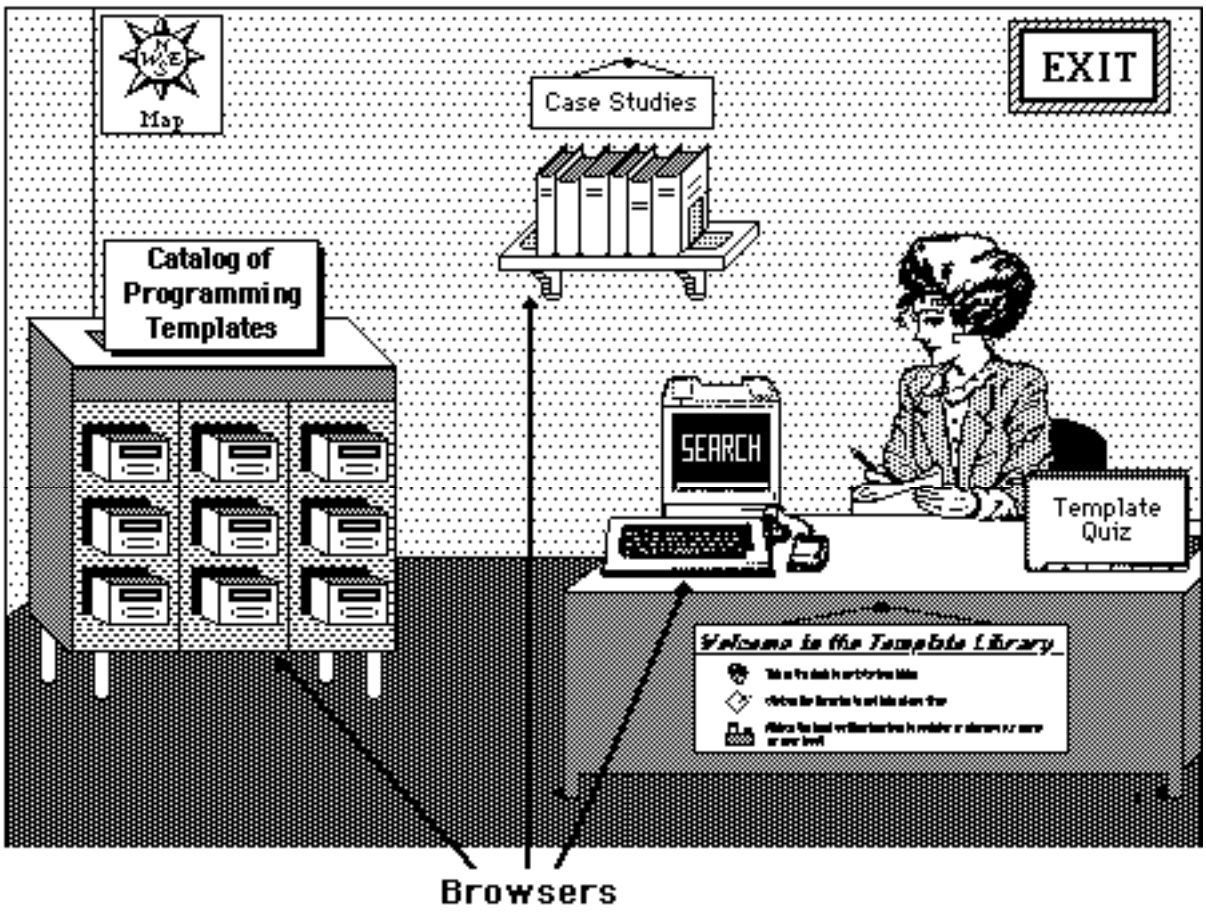


Figure 2. Public library interface metaphor.

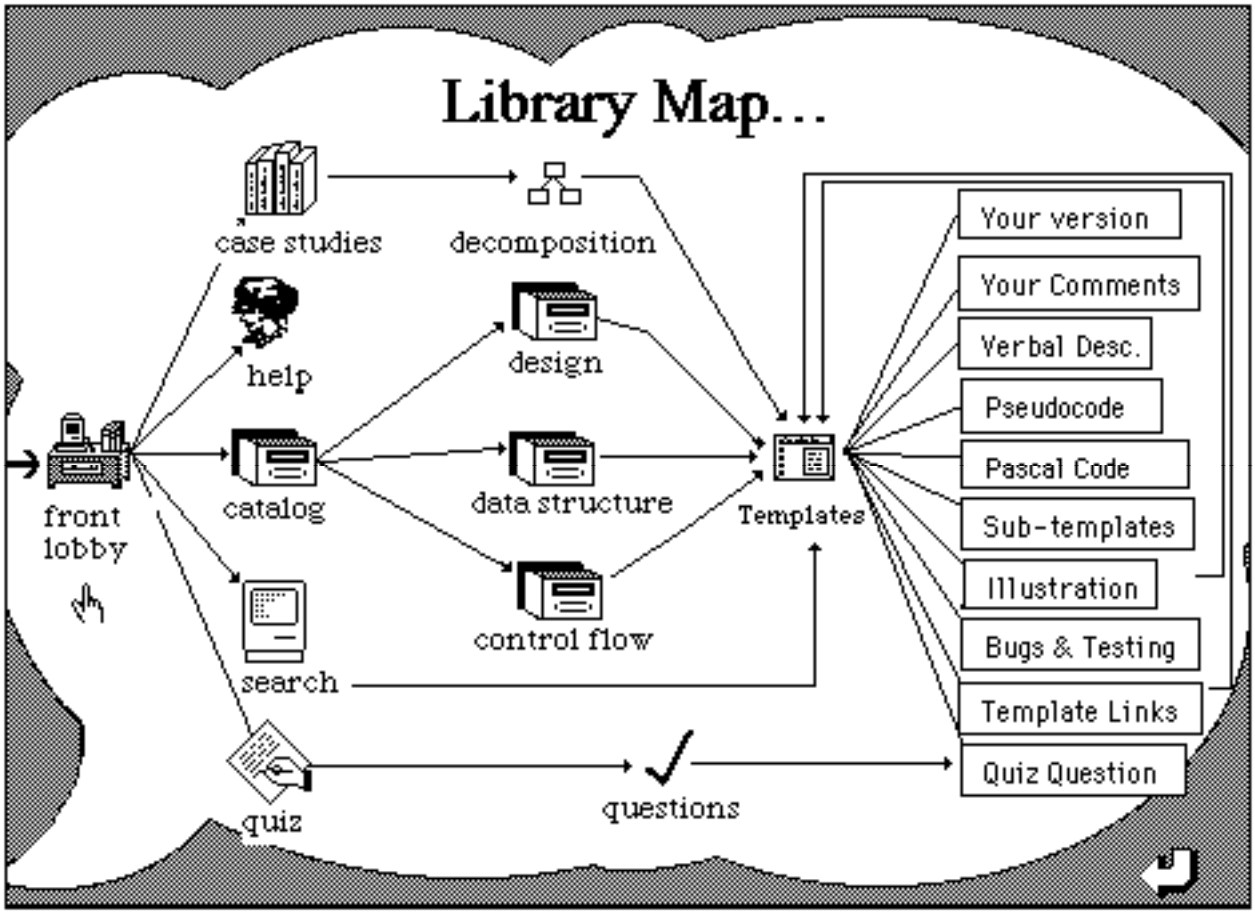


Figure 3. Template library map.

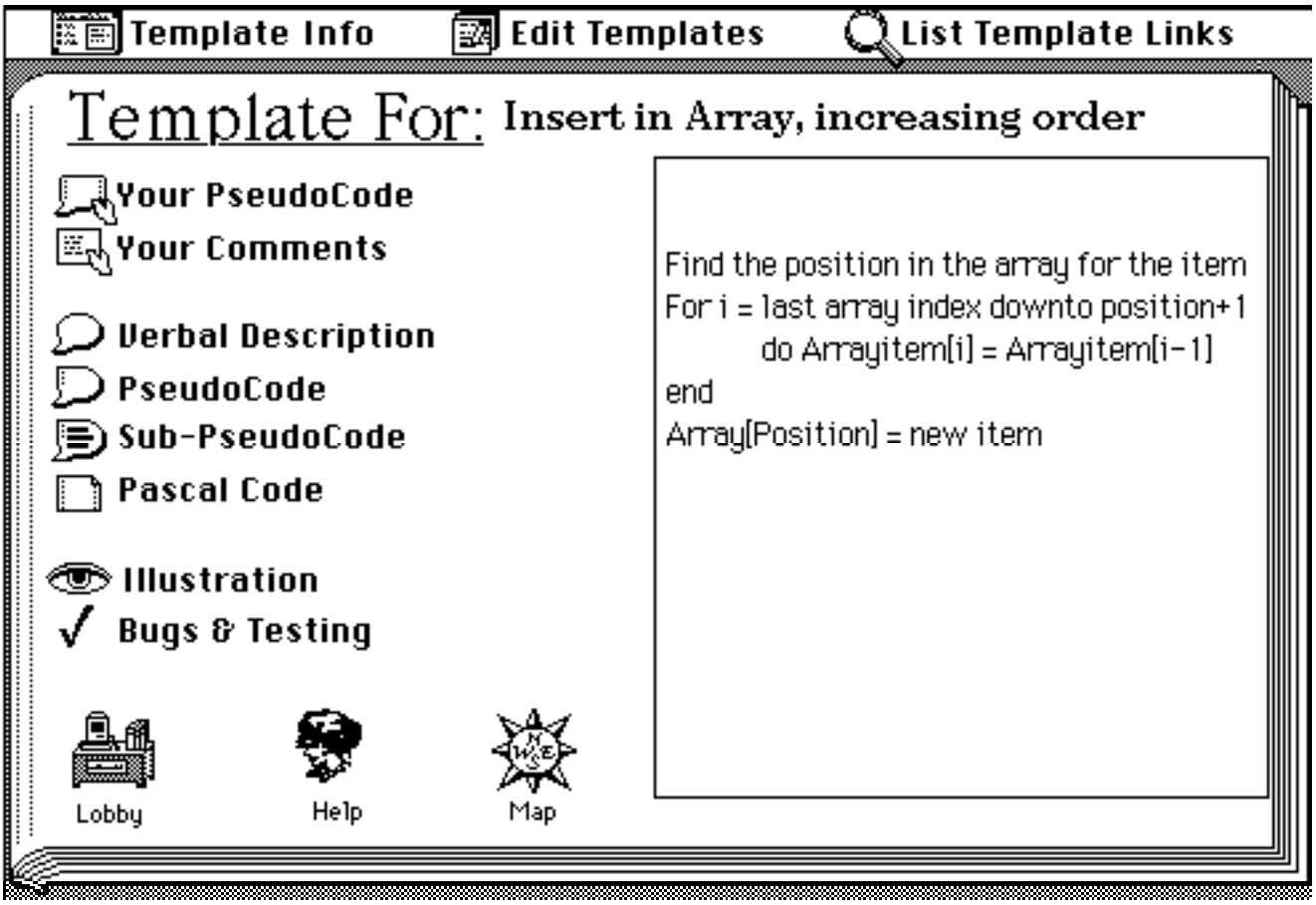


Figure 4. Insert into Array template.

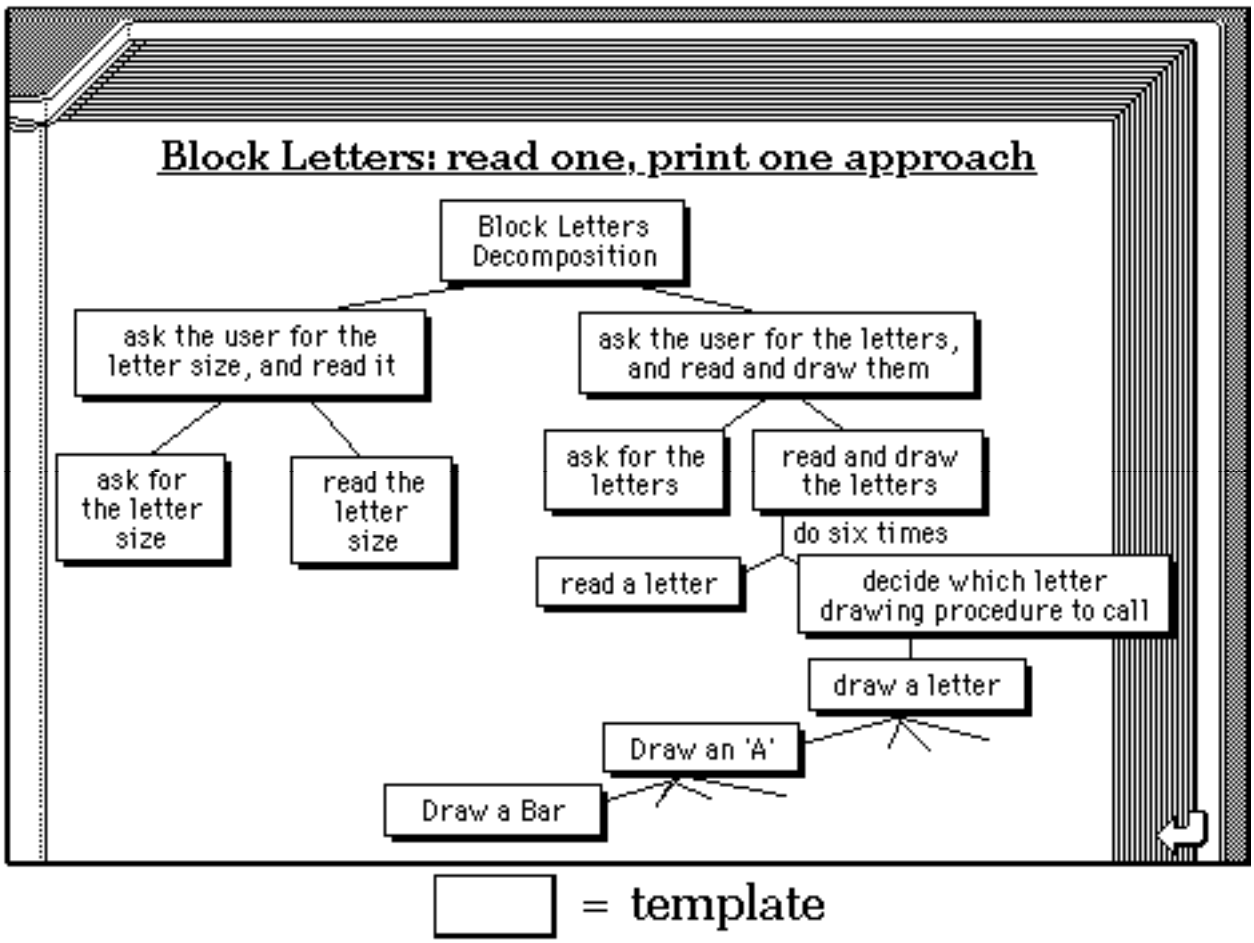


Figure 5. Block Letters Case Study decomposition diagram.

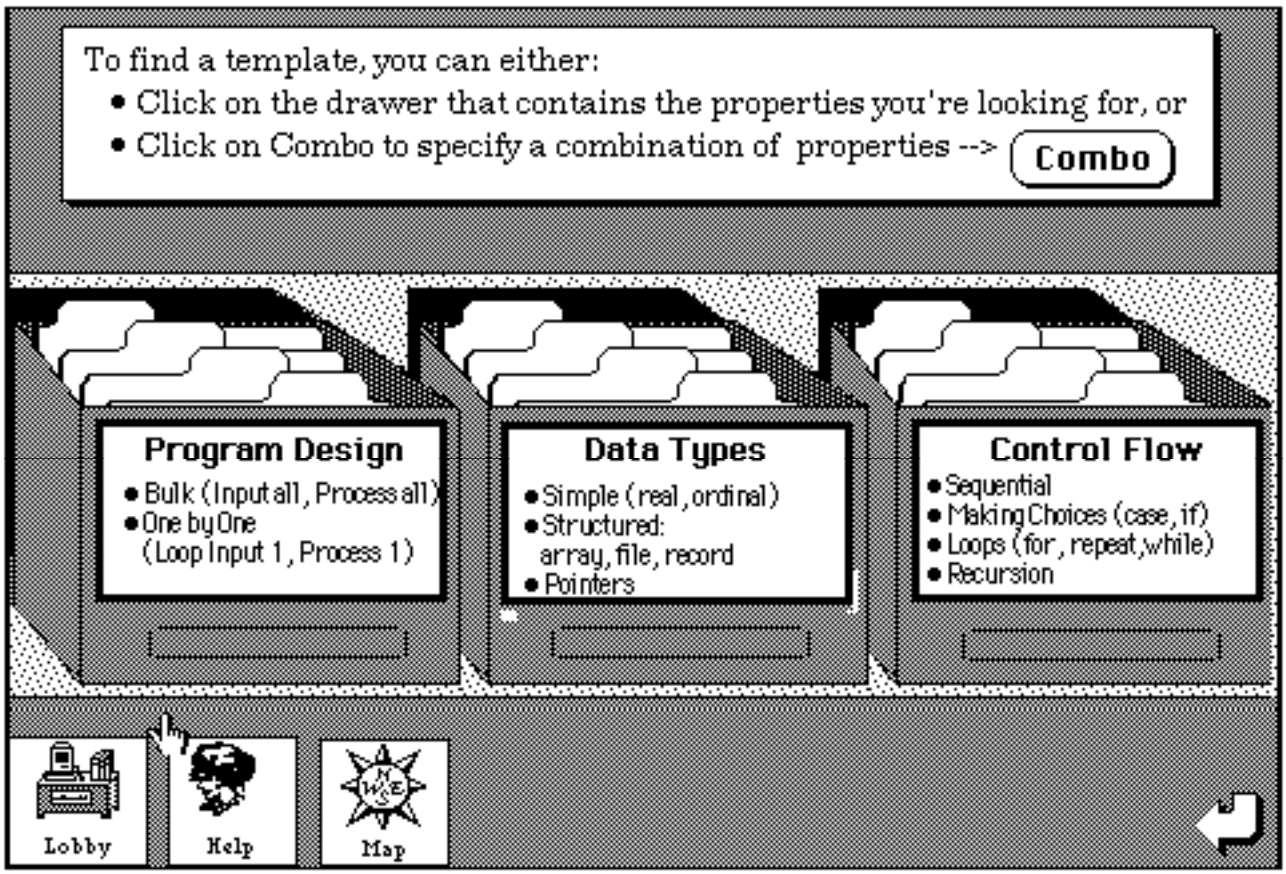


Figure 6. Catalog index of templates.

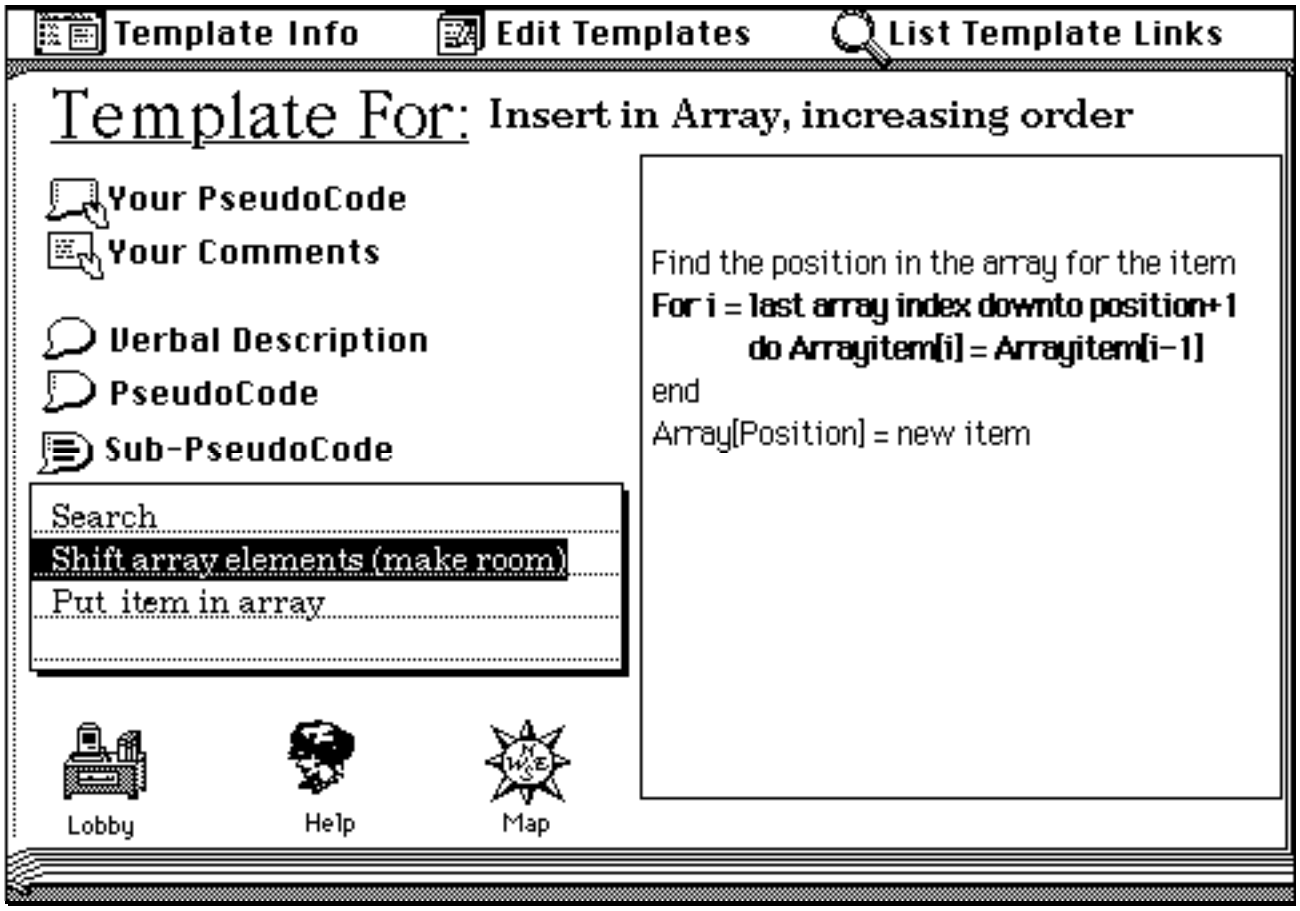


Figure 7. Sub-Pseudocode for Insert into Array template.

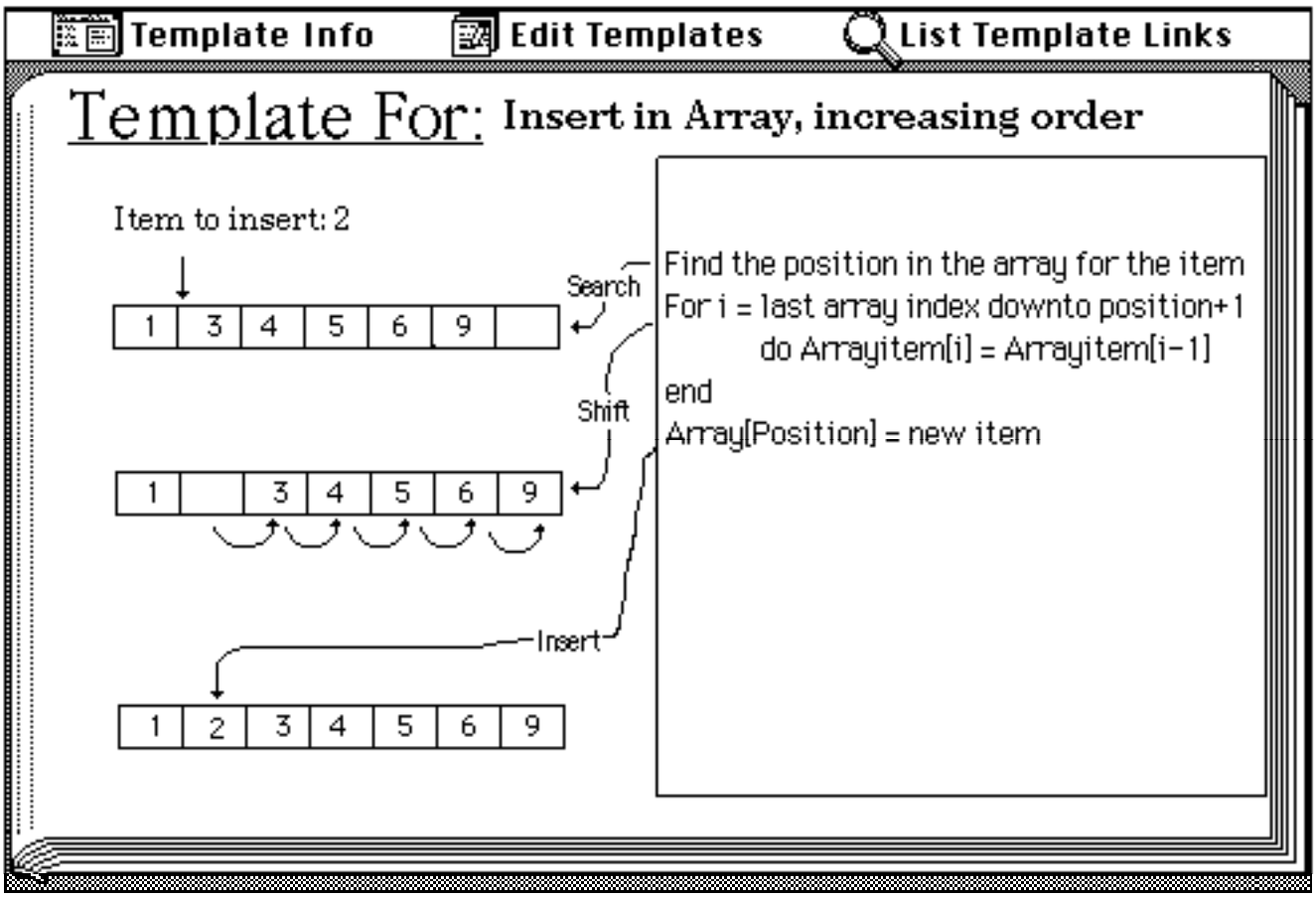


Figure 8. Illustration for Insert into Array template.