SPRUCE: A Framework for Software Restructuring

David Alex Lamb
David Putnam

February, 1991
External Technical Report
ISSN-0836-0227-
1989-244b

Department of Computing and Information Science
Queen's University
Kingston, Ontario K7L 3N6

Version 2.1
Document prepared Thursday, November 20, 1997

Abstract

Software restructuring can improve the maintainability and understandability of programs. We propose to divide restructuring into four stages: code restructuring, data restructuring, procedural restructuring, and remodularization. We survey prior restructuring work, and present our plans for SPRUCE, a framework for incorporating such work into an integrated tool.

Keywords and phrases: software restructuring, software maintenance, automated assistant

Computing Reviews categories:
D.2.7 Distribution and Maintenance - Restructuring
K.6.3 Software Management - software maintenance

General Terms: design, human factors

Table of Contents

List of Figures

1. Introduction

Software restructuring transforms a program’s source code into a more ‘‘structured’’ form, thereby improving its maintainability and prolonging its life. SPRUCE, a System for Providing assistance with RestrUcturing CodE, is an integrated collection of tools to assist a human in restructuring. Fully automatic restructuring is currently infeasible, and may remain so. This paper proposes to divide software restructuring into distinct types, and outlines an architecture and initial collection of facilities for SPRUCE, which we designed in late 1988 and began to implement in 1989-90.

1.1. Motivation

Software maintenance involves correcting (removing functional errors), adapting (enhancing), or perfecting (improving efficiency or performance) of software after its release for production use [1]. It can count for more than 50% of the costs in the lifetime of a software system [29]. Any technique that could reduce its cost is obviously valuable.

Maintenance is difficult because a maintainer must first understand the system before making changes; this currently accounts for about half of a maintenance programmer’s time [29]. The program is hard to understand if the original program was poorly written, or if previous maintenance degraded the program’s structure [28].

The structure of a program refers to both the structure of the code (called structured programming or programming-in-the-small) and the structure of the system (called module structure or programming-in-the-large [9]). Structured programming is an ‘‘art of reasoning’’ about the task, being able to abstract different levels of understanding of a problem, and connect them together in a hierarchy [18, 31]. Good module structure can be achieved by applying information hiding, in which every module hides a design decision, allowing the modules to be understood independently [30]. Although both these techniques were developed in the early 1970’s, some estimate that most programs in use today are unstructured [25, 35].

Good software structure can make the task of understanding the software much easier [2]. This can reduce the cost of maintenance to as little as one third of the cost of maintaining unstructured programs. Rewriting the unstructured programs from scratch using structured programming and information hiding techniques is impractical because of high costs [21].

Software restructuring is an alternative to software rewriting. However, this area has only recently begun to be studied, and further research is required [2]. This paper proposes a more detailed definition of restructuring (Section 1.2), reviews prior work on restructuring (Section 2), and outlines a framework for incorporating restructuring ideas into a common tool (Section 3).

1.2. Software Restructuring

Software restructuring creates an ‘‘equivalent structured replacement’’[34] from an unstructured source program. This can involve many types of transformations whose common goal is to make programs easier to understand and maintain. One way to make the study of restructuring easier is to divide it into smaller pieces, and view restructuring as a sequence of specialized restructuring stages: code level restructuring (Section 1.2.1), data restructuring (Section 1.2.2), procedural restructuring (Section 1.2.3), and remodularization (Section 1.2.4).

Each of these stages has three phases. *Information gathering* involves examining the input source program, compiling information about the program, and then presenting it to a restructurer (programmer or program) for analysis. *Decision making*, the most difficult phase, requires the restructurer to analyze the information and determine an appropriate structure for the software. *Execution* includes all the actions that implement the restructuring decisions.

1.2.1. Code Level Restructuring

Code level restructuring transforms program code to adhere to structured programming principles. Oulsnam [26] identifies several forms of code unstructuredness (often called ‘‘spaghetti code’’): jump into a decision, jump out of a decision, jump into the forward path of a loop, jump out of the forward path of a loop, jump into the backward path of a loop, and jump out of the backward path of a loop. These problems makes it difficult, if not impossible, to understand and maintain the code [11].

Code level restructuring is a logical first step in a complete restructuring since it greatly simplifies the following stages by allowing them to make additional assumptions about the source code.

1.2.2. Data Restructuring

Data restructuring makes the data structures and variable usage of the program more sensible. Data structure analysis includes making sure that all components of the data structures are related, that closely related data are not in separate structures, and that the best type of data structure is used. The data is much easier to understand if it is in a representation that abstracts its relevant similarities [16]. Variable analysis includes determining if variables are overloaded (that is, have two or more distinct roles), if a global variable should be local, and if variable parameters should be value parameters.

1.2.3. Procedural Restructuring

Procedural restructuring divides a program up into a logical set of routines. While many programs are already divided into routines, they are not necessarily the best possible divisions. Each routine should have only one entry point and one exit point, it should do a single abstract function, and the routines should be organized into a hierarchy [31]. Procedural restructuring may involve significant changes in the parameterization of routines, and may force further data restructuring.

1.2.4. Remodularization

Remodularization is the restructuring of an existing system into a modular hierarchy; it involves moving routines into appropriate modules. There are two distinct methods. Full remodularization involves the complete redesign of the modular structure. A designer deduces the requirements of the new system from the functionality of the old, then defines a new modular structure, independent of the existing structure. Maintainers then reorganize the source code into the new design. This method requires a large amount of effort all at once. Incremental remodularization involves examining the source code for recognizable information hiding modules and extracting them one by one. As more modules are recognized and extracted, the unstructured portion becomes smaller, making it easier to recognize additional modules [19]. This method amortizes the restructuring over a long time, but requires a larger total investment than for full remodularization, and the resulting module structure may not be as good.

2. Previous Research

High-level programming languages were first developed in the mid 1950’s [27]. Among the first of these were FORTRAN and COBOL, both of which are still widely used today. The main criteria for determining how good a program was were its speed and its size. This encouraged programmers to develop ‘‘tricks’’ and their own personal programming style. Throughout the 1960’s computers became faster, cheaper, and their memory capacity increased. Applications increased in difficulty and size, but programming styles remained the same. As programs grew larger, they became more and more difficult to understand. Dijkstra [10] attributed this to a lack of structure in programs, and sparked a debate over the use of the **goto** statement that continued for years. Knuth [18] outlines the history of this debate.

With the introduction and gradual acceptance of structured programming, newly developed programs became easier to understand. It was natural to try to find a way to get the benefits of structured programming out of the many unstructured programs previously written, without having to rewrite them.

2.1. Code and Procedural Restructuring

Bohm and Jacopini [5] showed it is possible to transform arbitrary flow diagrams into structured flow diagrams, sometimes by adding boolean variables. Ashcroft and Manna [3] extended this by introducing an algorithm for transforming arbitrary ‘‘goto programs’’ into equivalent ‘‘while programs.’’ Peterson et al. [32] showed how to transform any program into a ‘‘well-formed’’ program that only used if, repeat, and multi-level exit statements. Yourdon [37] introduced a boolean flag approach to eliminate multi-exit loops. Linger et al. [20] introduced a technique for parsing arbitrary flowgraphs into their prime components: sequence, if-then-else, and while-do.

deBalbine [7] used these ideas in his fully automatic FORTRAN ‘‘structuring engine’’ developed for Caine, Farber, and Gordon, Inc. It does not do any data restructuring or remodularization, but does some procedural restructuring if blocks of code appear more than once. It translates each subprogram into a flowgraph, transforms these into structured flowgraphs using node-splitting, then translates the result to S-FORTRAN, a superset of FORTRAN with structured constructs. Baker’s STRUCT program [4] at Bell Laboratories translates FORTRAN into RATFOR (another extended FORTRAN).

In the early 1980's the amount of work done in software restructuring continued to increase, particularly in the area of COBOL code and procedural restructuring. Lyons [21] and Miller [23] developed Structured Retrofit for the Catalyst Corporation. It makes programs more readable by removing all ALTER statements, eliminating procedure overlap caused by PERFORM THRU statements, eliminating some GOTOs by introducing PERFORMs, converting NOTEs to comments, eliminating control flow falling through one paragraph to the next, and removing unreachable code. To improve structure it creates an isolated control hierarchy, highlights looping conditions, places bounds on action modules, groups and standardizes all I/O, and consolidates all program termination to a single goback.

Also in 1981, Sage Software Products developed a COBOL restructurer to salvage a client's unmaintainable (but properly working) software system [6]. The restructurer generates a functionally equivalent version in structured pseudocode, which it then translates to COBOL. The total cost, including the restructurer, was less than 10% of the lowest estimate for rewriting the system.

Group Operations Inc. introduced another COBOL restructurer in 1984, called Superstructure [24]. This program evolved from SCAN/370, a COBOL static analyzer that helped maintainers understand unstructured COBOL programs. Superstructure placed all PROCEDURE DIVISION code in independent single entry/single exit procedures, eliminated GOTO statements except those to the beginning or exit point of the paragraph that contained them, removed all paragraph fall-throughs and ALTER statements, put unreachable code in comments, and eliminated PERFORM range violations.

Harandi [12] at the University of Illinois at Urbana-Champaign was more interested in the theoretical aspects of COBOL restructuring. His restructurer removes ALTER statements, replaces GOTO-DEPENDING-ON statements with IF-THEN-ELSE statements, transforms all procedures into single entry/single exit routines, transforms unstructured flows of control to structured equivalents, and simplifies complex control structures by simulating them with disciplined uses of GOTO statements. This work also provides experimental results showing the improved maintainability of restructured programs.

2.2. Data Restructuring and Remodularization

There has been little prior work on data restructuring or remodularization. The Leeds Transformation System, developed in 1983 at the University of Leeds [22], handles some data restructuring. It removes redundant variables and assignments, moves loop invariant statements, and collapses implicit loops. Since Parnas' paper on information hiding [30], most of the work on modularity has dealt only with designing new software or completely rewriting existing systems (such as the redesign of the software for the A-7 aircraft [15]). Lamb [19] has proposed incremental remodularization, and Arnold [2] has recognized the need for more research in this area.

3. A Restructuring Tool

This section describes the System for Providing assistance with RestrUcturing CodE (SPRUCE). SPRUCE is a collection of user-visible facilities designed to help a restructuring programmer in carrying out each of the restructuring stages of Section 1.2. SPRUCE supplies all the information required by each stage, assists in the decision making, and makes the execution of the restructuring decisions less error prone. SPRUCE also lays a foundation for future enhancements, perhaps leading to fully automatic restructuring.

3.1. Language Dependence

Most of the SPRUCE facilities depend on the source language of the system it is restructuring. An important part of making SPRUCE flexible is to minimize its language dependencies; one of our few assumptions is some way to split a program into an interface file (for externally visible declarations) and a program file. Thus SPRUCE will contain several modules with a language-independent algorithm, and tables or parameters that specialize them for the particular source language.

Our main plan is to use abstract syntax trees to represent the source program. The type of tree nodes available may differ from language to language, but they overlap. For example, the concrete syntax of the if statements in C and Pascal (Figure 1) are different, yet we could use the same node type to represent both.

Concrete Syntax

Pascal:

if *boolean-expression*
then *statement*
[else *statement*]

C:

if (*expression*)
statement
[else *statement*]

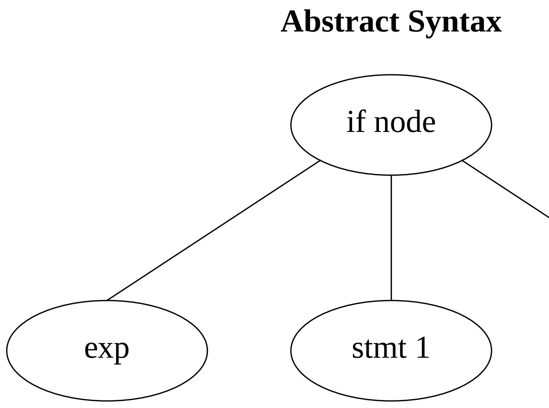

Figure 1: Comparison of IF Statements in C and Pascal

To limit the language dependencies, SPRUCE processes the tree with a language-independent tree walker. The tree walker understands the structure of the abstract syntax tree and provides routines to allow access of the information in the nodes and to instruct the tree walker to move between nodes. Table entries for the each node type tell the tree walker what to do. Figure 2 describes an example trees walker in pseudo-code to provide a more concrete description.

3.2. Philosophy

Designing an automatic restructuring tool is in many ways similar to designing an automatic natural language translation tool. Both require specialized human talents and neither is understood nor even defined well enough for automatic tools to be feasible. However, Kay [17] proposes the ‘‘Translator’s Amanuensis,’’ which implements only well understood areas of language translation, like making vocabulary suggestions from a dictionary. The human expert solves the less understood problems, like choosing an exact word based on context, just as she would without the tool. The Amanuensis should be flexible enough to include any parts of language translation that may be solved in the future.

Our work parallels this proposal. SPRUCE concentrates on known areas of software restructuring, and focuses particularly on the information gathering and execution phases of each stage. The decision making phases are still not well understood, and so are left to the human expert. It is possible to add new features to SPRUCE as they become better understood.

The following sections describe several facilities that should be useful in restructuring, along with how to apply them to the various stages of software restructuring. These

```
routine TreeWalker with parameter Table
     initialization code
     call ProcessNode(Table,RootNode)
     ending code
end TreeWalker

routine ProcessNode with parameters Table, NodeID
     call Table[NodeType(NodeID)]
     for all subnodes of NodeID
          call ProcessNode(Table, Subnode)
end ProcessNode
```

Figure 2: Pseudo-Code Tree Walker

sections describe how a restructuring programmer may use the facilities, but avoid describing precise user interfaces (which are subject to change following feedback from use). Figure 3 summarizes the facilities we propose, showing what phase (information gathering or execution) and stage (code, data, procedural, or remodularization) they are part of, what inputs they take, and what outputs they produce. To save space we avoid replicating this information in the individual sections.

3.3. Call Graph Analysis

This facility generates and displays the call graph of the program. After SPRUCE analyses the input source code, the user may request display of the call graph in either graphical or textual form. The information can be used in full remodularization to help determine which procedures will go into the new modules, and by incremental remodularization to help recognize potential modules. There are standard techniques for creating a call graph [14] and displaying graphs in textual form [8, 36].

3.4. Code Shifter

This facility allows the programmer to transfer routines and data from the old system to the new one. The user can select a procedure and specify its visibility and what module

Facility	Phase	Stage(s)	Input	Output
Call Graph Analysis	Info	Proc/ Remod	source code	call graph
Code Shifter	Exec	Remod	source, module structure diagram, data flow diagram, call graph	updated source
Data Flow Analysis	Info	Data	any source	data flow diagram, variables for further study
Goto Analysis	Info/ Exec	Code	original source	improved source
Interface Controller	Exec	Remod	source, module structure diagram	updated source
Module Structure Controller	Exec	Remod	nothing, or old module structure diagram	new module structure diagram
Procedural Analysis	Info	Proc	data restructured source	list of procedures to restructure
Procedural Rewriting	Exec	Proc	data restructured source, call graph	procedure restructured source
Structure Organizer	Exec	Full Remod	module structure diagram	directory with skeletal files
Variable Renaming	Exec	Data	any source code	renamed source code
Variable Usage Analysis	Info	all but Code	source, data flow diagram	information requested by user

Figure 3: Summary of Facilities

to move it to. SPRUCE examines the call graph and data flow diagram to determine if the module structure diagram will be violated.

The user may also select variables and data types to move; SPRUCE lists all their uses to help find routines that may also need to move. If the user does not want to move all the routines that access the data, he can create new routines to allow other modules to set and fetch values that used to be global.

3.5. Data Flow Analysis

This facility creates and analyzes a data flow diagram to help user select candidate variables for further analysis and possible restructuring. The user may request display of the data flow diagram in either graphical or textual form, and may also request further analysis of the data flow diagram to identify unstructured data and variable usage. Criteria for this include: how widespread a variable's usage is; what values are never possible in certain contexts; which variables are changed and then never used; which variables are mostly used as counter or index variables. Before any actions are taken, the results should be analysed using the variable usage facility described below. There are standard techniques for creating a flow graph [14] and displaying graphs in textual form [8, 36].

3.6. Goto Analysis

This facility analyzes programs for harmful uses of **goto** statements, and replaces them as the user requests. It shows the context of each **goto** statement, labeled as harmful or harmless appropriately. Harmful uses of the **goto** are those that create Oulsnam's unstructured forms [26], or cause abnormal exit or entry of procedures. Harmless uses simulate higher level constructs, such as transferring control to the end of a loop [7]. For each **goto** SPRUCE asks the user whether to remove it. For a description of replacing **goto**s with structured programming techniques, see Ashcroft and Manna [3].

3.7. Interface Controller

This facility helps change routines and types in a module from visible to hidden, or vice versa. The user can invoke this facility on any program or interface file in the remodularization directory. In a program file the user may select a hidden (that is, non-interface) type or routine, and change it to visible; this creates a corresponding entry in the interface file. In an interface file the user may select any visible type or routine and change it to hidden. Before complying, SPRUCE checks the modules that depend on this module (according to the module structure diagram) to determine if any of them use the type or routine selected. If any do, it tells the user and avoids the change. Otherwise, it removes the declaration from the interface file.

3.8. Module Structure Controller

This facility keeps track of the module decomposition and dependencies. The user may edit the structure by adding new modules and dependencies, or by deleting existing modules or dependencies. At the end of the session SPRUCE checks that the module structure remains sensible. It forbids disconnected components, requires a single root, and issues warnings for each cycle in the graph. For full remodularization the user enters the new modular design at the beginning of remodularization; for incremental remodularization she starts with the original decomposition.

3.9. Procedural Analysis

This facility helps the restructuring programmer decide what procedures to rewrite. It applies software metrics to each procedure, then displays procedure names with corresponding metric information. Users may enter threshold values for each metric to mark the boundary between acceptable and unacceptable procedures. Harrison et al. [13] survey metrics.

3.10. Procedural Rewriting

This facility makes rewriting procedures easier and less error prone. Depending on the state of the procedure being rewritten and the intentions of the restructuring programmer there are two types of procedural rewriting: refinement and splitting. After procedural restructuring, it may be good to do data restructuring again, particularly if many new routines have been created.

Refinement restructures procedures that have not been decomposed far enough. The user selects a procedure, names a new procedure, and selects sections of code from the original procedure to move to the new one. SPRUCE examines the new routine’s variables; parameters of the original routine become parameters of the new routine; locals used only in the new routine become locals of the new routine; remaining locals become parameters. SPRUCE then places an appropriate call to the new routine in the parent routine.

Splitting replaces procedures that do several distinct functions by several smaller routines, one for each distinct task. Splitting begins like refinement, deletes the original routine, then (for each call of the original routine) asks the user which new procedure to call instead of the original.

3.11. Structure Organizer

This facility sets up a skeleton module structure. Using the dependencies and module names in the module structure diagram, SPRUCE creates a command file (or UNIX makefile) for compiling the new version, and an empty interface file and program file for each module. Each program file will contain statements linking it to its own interface file as well as the interface files for each module it depends on directly. This is only done at the beginning of full remodularization to set up a framework.

3.12. Variable Renaming

This facility helps change variable names without changing the meaning of the program. The user selects any occurrence of a variable and enters a new name. SPRUCE examines all other occurrences to see if the scope rules change the meanings of the statements. For example, in Pascal, if a record field name is changed to a name already in use by a global variable (which is legal), any **with** statement for that record will have to be checked to insure that any intended accesses of the global variable are not captured.

3.13. Variable Usage Analysis

This facility traces variables' potential data flow through the system, allowing the user to determine if their usage is consistent and appropriate. It displays the source code, allowing the user to select any occurrence of a variable to analyze, then shows related occurrences by following the data flow. It can be used in procedural restructuring to determine how a set of variables are related, and in remodularization to help determine closely related routines by finding which have the most variables in common.

4. Conclusion

We have defined several software restructuring stages, and have outlined a collection of facilities (based on research of many other people) that can fit together into a coherent tool. We invite others to propose additional facilities that might fit into the same framework.

So far we have said little about how feasible it would be to build SPRUCE. It should be clear from the descriptions in Section 3 that most of the individual facilities are straightforward applications of well-understood work. Furthermore, Putnam [33] has done a detailed design of a carefully chosen subset of the SPRUCE facilities, to the level where an experienced programmer could implement them.

Since individual facilities are of medium size and independent of each other, it is possible to build SPRUCE incrementally. To date we have constructed

- A Pascal parser, pretty-printer, and partial semantic analyser (for name resolution).
- Implementations of lines-of-code, Halstead, McCabe, and a few lesser-known metrics.
- Partial implementation of a SUNView-based user interface for browsing through Pascal source code.
- Data- and control-flow analysis.
- Implementation of the "variable renaming" facility.

These pieces are not yet all integrated with each other; they arose as separate student projects.

Acknowledgements

Katherine Franklin built the Pascal parser and pretty-printer. Heather Burles did the name resolver, the metrics, and a first version of the flow analyser. Wendy Sharp did the SUNView interface. Mary-Ellen Maybee revised the flow analysis and built the variable renaming facility.

Margaret Lamb gave helpful comments on an earlier draft of this paper. This work was supported in part by the Natural Sciences and Engineering Research Council

(NSERC) of Canada under Grant OPG0000908, in part by the NSERC graduate student fellowship program, and in part by the Information Technology Research Centre (ITRC), which is part of the Ontario Centres of Excellence program.

References

1. Robert S. Arnold and Donald A. Parker, ‘‘The Dimensions of Healthy Maintenance,’’ in *Proceedings of the Sixth International Conference on Software Engineering*, pages 10-27, IEEE Computer Society (1982).
2. Robert S. Arnold, ‘‘An Introduction to Software Restructuring,’’ in *Tutorial on Software Restructuring*, pages 1-11, IEEE Computer Society (1986).
3. Edward Ashcroft and Zohar Manna, ‘‘The Translation of ‘Goto’ Programs to ‘While’ Programs,’’ in C. V. Freiman, editor, *Proceedings of the IFIP Congress 71, vol. 1*, pages 250-255, North-Holland Publishing Company (1972).
4. Brenda S. Baker, ‘‘An Algorithm for Structuring Flowgraphs,’’ *Journal of the ACM* **24**(1):98-120 (January 1977).
5. Corrado Bohm and Guiseppe Jacopini, ‘‘Flow Diagrams, Turing Machines and Languages with Only Two Formation Rules,’’ *Communications of the ACM* **9**(5):366-371 (May 1966).
6. Kevin Burns, ‘‘Using Automated Techniques to Improve the Maintainability of Existing Software,’’ in *DSSD User’s Conference/6-Maintenance*, pages 33-39 (1981).
7. Guy deBalbine, ‘‘Better Manpower Utilization Using Automatic Restructuring,’’ in *AFIPS Conference Proceedings, vol. 44*, pages 319-327 (1975).
8. Guy deBalbine, ‘‘MTR - A Tool for Displaying the Global Structure of Software Systems,’’ in *AFIPS Conference Proceedings, vol. 47*, pages 571-580 (1978).
9. F. DeRemer and H. H. Kron, ‘‘‘Programming-in-the-Large’ vs. ‘Programming-in-the-Small’,’’ *IEEE Transactions on Software Engineering* **SE-2**(2):80-86 (June 1976).
10. Edsger W. Dijkstra, ‘‘Go to Statement Considered Harmful,’’ *Communications of the ACM* **11**(3):147-148 (March 1968).
11. Barnaby J. Feder, ‘‘Straightening Out the Spaghetti Code,’’ *The New York Times*, :F5 (May 8, 1988).
12. M. T. Harandi, ‘‘An Experimental COBOL Restructuring System,’’ *Software: Practice and Experience* **13**(9):825-846 (September 1983).
13. Warren Harrison, Kenneth Magel, Raymond Kluczny, and Arlan DeKock, ‘‘Applying Software Complexity Metrics to Program Maintenance,’’ *IEEE Computer* **15**(9):65-79 (September 1982).
14. Matthew S. Hecht, *Flow Analysis of Computer Programs*. Elsevier North-Holland, Inc (1977).

15. Kathryn L. Heninger, ‘‘Specifying Software Requirements for Complex Systems: New Techniques and Their Applications,’’ *IEEE Transactions on Software Engineering* **6**(1):2-13 (January 1980).
16. C. A. R. Hoare, ‘‘Notes on Data Structuring,’’ in *Structured Programming*, pages 83-174, Academic Press (1972).
17. Martin Kay, ‘‘The Proper Place of Man and Machines in Language Translation,’’ Technical Report CSL-80-11, Xerox Palo Alto Research Center (October 1980).
18. Donald E. Knuth, ‘‘Structured Programming with goto Statements,’’ *Computing Surveys* **6**(4):261-301 (December 1974).
19. David Alex Lamb, *Software Engineering: Planning for Change*. Prentice-Hall, Inc. (1988).
20. Richard C. Linger, Harlan D. Mills, and Bernard I. Witt, *Structured Programming: Theory and Practice*. Addison-Wesley (1979).
21. Michael J. Lyons, ‘‘Salvaging Your Software Asset (Tools Based Maintenance),’’ in *AFIPS Conference Proceedings, vol. 50*, pages 337-341 (1981).
22. B. Maher and D. H. Sleeman, ‘‘Automatic Program Improvement: Variable Usage Transformations,’’ *ACM Transactions on Programming Languages and Systems* **5**(2):236-264 (April 1983).
23. Jon Cris Miller, ‘‘Structured Retrofit,,’’ in Girish Parikh, editor, *Techniques of Program and System Maintenance*, pages 181-182, Little, Brown, and Company (1982).
24. Henry W. Morgan, ‘‘Evolution of a Software Maintenance Tool,’’ in *Proceedings of the Second National Conference on EDP Software Maintenance*, pages 268-278 (1984).
25. Henry Mullish, *Structured COBOL: A Modern Approach*. Harper & Row Publishers (1983).
26. G. Oulsnam, ‘‘Unraveling Unstructured Programs,’’ *The Computer Journal* **25**(3):379-387 (August 1982).
27. Donald E. Knuth and L. T. Pardo, ‘‘The Early Development of Programming Languages,,’’ in J. Belzer, A. G. Holtzman, and A. Kent, editors, *Encyclopedia of Computer Science and Technology, vol. 6*, pages 419-493, Marcel Dekker (1977).
28. Girish Parikh, ‘‘The World of Software Maintenance,’’ in *Techniques of Program and System Maintenance*, Little, Brown, and Company (1982).
29. Girish Parikh and Nicholas Zvegintzov, ‘‘Introduction to Tutorial on Software Maintenance,’’ in *Tutorial on Software Maintenance*, pages ix-xi, IEEE Computer Society (1983).
30. David L. Parnas, ‘‘On the Criteria to be Used in Decomposing Systems Into Modules,’’ *Communications of the ACM* **15**(12):1053-1058 (December 1972).

31. David L. Parnas, ‘‘On a Buzzword: Hierarchical Structure,’’ in *Proceedings of IFIP Congress 74*, North Holland (1974).
32. W. W. Peterson, T. Kasami, and N. Tokura, ‘‘On the Capabilities of While, Repeat, and Exit Statements,’’ *Communications of the ACM* **16**(8):503-512 (August 1973).
33. David W. Putnam, *Software Restructuring*, M.Sc. dissertation, Queen’s University Department of Computing and Information Science (February 1989).
34. Donald J. Reifer and Stephen Trattner, ‘‘A Glossary of Software Tools and Techniques,’’ *IEEE Computer* **10**(7):52-60 (July 1977).
35. Norman F. Schneidewind, ‘‘The State of Software Maintenance,’’ *IEEE Transactions on Software Engineering* **SE-13**(3):303-310 (March 1987).
36. Ben Shneiderman, Philip Shafer, Roland Simon, and Linda J. Weldon, ‘‘Display Strategies for Program Browsing: Concepts and Experiment,’’ *IEEE Software* **3**(3):7-15 (May 1986).
37. Edward Yourdon, *Techniques of Program Structure and Design*. Prentice-Hall, Inc. (1975).