Software Restructuring

ROBERT S. ARNOLD

Invited Paper

Perhaps the most common of all software engineering activities is the modification of software. Unfortunately, software modification—especially during software maintenance—often leaves behind software that is difficult to understand for those other than its author. The result is software that is harder to change, less reliable when it is changed, and progressively less likely to be changed. Software restructuring is a field that seeks to reverse these effects on software.

This paper is a brief tutorial on software restructuring. The paper discusses what restructuring is, advantages and disadvantages of restructuring, tools and case studies, and future possibilities. The reader is assumed to have a general appreciation for building and maintaining software systems. After reading this paper, the reader should have a feel for the strengths, weaknesses, and capabilities of software restructuring technology.

I. Introduction

A. The Concept of Software Restructuring

Software restructuring is the modification of software to make the software easier to understand and to change, or less susceptible to error when future changes are made. "Software" includes external and internal documentation concerning source code, as well as the source code itself.

This definition of software restructuring excludes software changes for other purposes, such as code optimization. Code optimization does imply "restructuring" in a sense, but normally does not concern the key element—for this paper—of improving software maintainability.

Some examples of software restructuring are pretty printing (spaces and aligns software statements so that they are easier to comprehend as logical units); manual restructuring according to coding style standards (infuses software with standard, recognizable structure); editing documentation for readability (makes the documentation easier to read, possibly improving software maintainability); creating an index for some software documentation (makes it easier to locate information about software); and so on.

Fig. 1 illustrates software restructuring. The original program is hard to maintain. Its purpose is apparently to edit some cost indicators (see line 100), but other details, such as the meaning of the cost indicators being edited, are unclear. Its variables are not mnemonic. (What does "SPAG" mean in line 140?) More than one statement per line, lack of consistent indentation, and splitting of statements across physical lines makes following the control flow difficult. The program has literal constants (e.g., "007"), instead of symbolic constants (e.g., SPECIAL-AGENT with a constant value of "007"), which makes the program harder to change.

ORIGINAL PROGRAM FRAGMENT:

```
000100 EDIT-COST-INDICATORS.
000110    IF C NOT EQUAL TO "A" OR "B" MOVE "X" TO COST-INDICATOR
000120       GO TO EDIT-COST-INDICATORS-EXIT.
000130                IF C EQUAL TO "A" AND CS EQUAL TO 1 MOVE 0 TO
000140                    PGNT ADD1 TO PGNT MOVE "007" TO SPAG.
000150    MOVE ALL NINES TO ZCOD ADD 1 TO NLCNT
000160    ADD SPC TO CUM-SPC PERFORM OK-RECD-PRINT THROUGH
000170       OK-RECD-SUB-PRINT GO TO EDIT-COST-INDICATORS-EXIT.
000180     IF C EQUAL TO "B" AND CS EQUAL TO 1 MOVE 0 TO PGNT ADD 1
000190           TO PGNT MOVE "007" TO SPAG     MOVE "99999" TO ZCOD,
000200                ADD 1 TO NLCNT                ADD SPC TO CUM-SPC
000210       PERFORM OK-RECD-PRINT THROUGH OK-RECD-SUB-PRINT
000220    GO TO EDIT-COST-INDICATORS-EXIT.
000230             IF C EQUAL TO "A" OR "B" AND CS NOT EQUAL TO 1 ADD 1
000240          TO PGNT MOVE "010" TO SPAG     ADD 2 TO NLCNT
000250        ADD SPC TO CUM-SPC PERFORM NOT-OK-RECD-PRINT.
000260 EDIT-COST-INDICATORS-EXIT.
000270  EXIT.
```

RESTRUCTURED PROGRAM FRAGMENT:

```
000100 EDIT-COST-INDICATORS-1080.
000110    IF COST-INDICATOR = "A" OR "B",
000120       IF SUB-COST-INDICATOR = 1,
000130          MOVE 0 TO PAGE-COUNT,
000140          ADD 1 TO PAGE-COUNT,
000150          MOVE "007" TO SPECIAL-AGENT,
000160          MOVE "99999" TO ZIP-CODE,
000170          ADD 1 TO NEW-LINE-COUNT,
000180          ADD SPECIAL-COST TO CUMULATIVE-COST,
000190          PERFORM OK-RECD-PRINT-1470 THROUGH
000200             OK-RECD-PRINT-1580,
000210       ELSE
000220          ADD 1 TO PAGE-COUNT,
000230          MOVE "010" TO SPECIAL-AGENT,
000240          ADD 2 TO NEW-LINE-COUNT,
000250          ADD SPECIAL-COST TO CUMULATIVE-COST,
000260          PERFORM NOT-OK-RECD-PRINT-1780,
000270    ELSE
000280       MOVE "X" TO COST-INDICATOR.
000290 EDIT-COST-INDICATORS-EXIT-1260.
000300    EXIT.
```

Fig. 1. Software restructuring of a COBOL program. (From [25], p. 129.)

The restructured program has fixed many of these problems. From quick inspection, the purpose of the program has something to do with accumulating costs related to special agents. (James Bond's—agent 007's—expense accounting program?) Statements appear on one line. The control flow has been simplified and indentation makes the control

Manuscript received August 8, 1988; revised October 18, 1988. In preparing this paper, the author has reused and updated much of the material in [5]. All views are the author's only, and do not necessarily reflect those of the Software Productivity Consortium.

The author is with the Software Productivity Consortium, Herndon, VA 22070 USA.

IEEE Log Number 8926704.

flow easy to follow. Variables have more meaningful names (e.g., SPECIAL-AGENT instead of SPAG). Literal constants like "007" still remain, but their purpose is easier to fathom (e.g., since "007" is assigned to SPECIAL-AGENT, 007 appears to be the number of a special agent).

Several terms related to software restructuring have appeared. Software reengineering, software renewal [51], software renovation, software rejuvenation [16], [17], software improvement [25], [26], software recycling [52], and so on, have approximately the same intent as software restructuring: modifying or adding to software to make it easier to understand and to change.

Reverse engineering is the recovery of information about software to make the software easier to understand and to change. For example, creating design diagrams for a system whose design diagrams are out of date is a reverse engineering technique.

Reverse engineering and software restructuring are related. Reverse engineering connotes adding new information to software where such information previously did not exist, or was hopelessly inaccurate. Software restructuring connotes taking existing information and refashioning it so it can be more easily understood. However, since the process of refashioning information often incorporates new insights about software, for this paper we consider reverse engineering a part of software restructuring.

B. Why Be Concerned About Software Restructuring?

With continued change, software tends to become less "structured" [31]. This is manifested by out-of-date documentation, code which does not conform to standards, increased time for programmers to understand code, increased ripple effect of changes, and so on. These can—and usually do—imply higher software maintenance costs.

Software restructuring is an important option for putting high software maintenance costs under control. The idea is to modify software—or programmer's perceptions of software structure—so one can understand and control it anew.

There are many other reasons why software engineers should be aware of software restructuring:

- Regaining understanding of software by instilling software with known, easily traceable structure. This has the side benefits of
  - easier documentation,
  - easier testing,
  - easier auditing,
  - potentially reduced software complexity,
  - potentially greater programmer productivity,
  - reduced dependence on individuals who alone understand poorly structure software,
  - increased interchangeability of people maintaining software, and
  - greater programmer job satisfaction due to decreased frustration in working with poorly structured software.
- Creating software whose structure more closely resembles the structure taught to newer generations of programmers
- Reducing the amount of time needed for maintenance programmers to become familiar with a system
- Upgrading software along with upgrading software engineering practices
- Implementing standards for software structure
- Making bugs easier to locate
- Extending system lifetime by retaining a system's flexibility through good structure
- Preparing software as input for software analysis tools
- Preparing software for conversion
- Preparing to add new features to software
- Preserving software's asset value to an organization

Software restructuring is an integral part of achieving many goals in software maintenance and in corporate planning for software change.

C. What "Structure" is the Target of Restructuring?

Software restructuring is not just concerned with objectively observable/measurable aspects of software structure. It is also concerned with people's perceptions of software structure. When asked about the understandability of a piece of software, most software engineers will not base their answer exclusively on measurements of objective software aspects; they also look at at least part of the software.

The point here is that in considering restructuring, one needs to consider that the structure (ideas about the software) in people's heads matters as much as the objective structure of software itself. For software restructuring, "structure" is determined by at least two things: the software and the perceiver. Thus anything that can influence the software's state or the perceiver's state might influence software structure. The succeeding discussion will use the term "software structure" with both these aspects of structure in mind.

Fig. 2 presents some of the factors influencing people's perceptions of software structure. The clearest influence

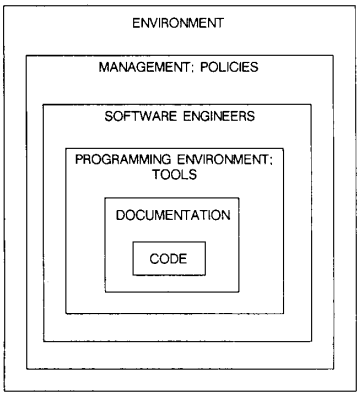

Fig. 2. Factors influencing perceptions of software structure.

on software structure comes from the software code itself, at the core of Fig. 2. Next, the in-line software documentation—often the primary documentation programmers depend on [22]—can have a strong influence on a programmer's perception of structure. Other documentation, when available, up-to-date, easily referenced, clearly written, and so on, can also influence a programmer's perception of structure. Next come the set of available software

tools (or programming environment) which can illuminate different views of the software for the programmer. For example, program traces can show dynamic execution behavior, algorithm animation can help programmers understand the dynamic strategy behind an algorithm, global variable cross referencers can help programmers understand the interactions between modules, and pretty printers can make reading code much more appealing.

In the middle layer of Fig. 2 is the programmer, the "other side" of software structure. If the programmer is not trained to look for certain aspects of structure, his or her perception of structure will be influenced.

Because programmers' perceptions differ (even in the same programmer at different times[1]), the notion of software structure is not constant, but *dynamic*. In fact, failure to have written notions of structure (such as structural standards) which programmers are aware of, or failure to allow programmers to refresh their notions of structure by viewing software in various ways, can lead to a net *loss* of structure. For example, program patches are insidious structure-reducers because the reason for the patch is often lost when the patch's author leaves. In effect, the programmer has promoted in the software his or her structural view without telling others of this view, resulting in a net loss of software structure for others.

At the next higher layer of Fig. 2 is the management layer, which can sensitize a programmer to aspects of software structure. For example, management can emphasize software quality as embodied in a set of standards. If a programmer's performance review is tied to how well these standards are met, then the programmers' perceptions of software will likely be influenced!

The highest layer of Fig. 2 is the environment surrounding the lower levels. This includes physical facilities, degree of influence on software maintenance tasks by computer users, lack of availability of hardware, and so on. All can influence a programmer's attitude, which in turn can influence his or her perception of software structure.

By changing the state of items at any of the levels in Fig. 2, "software structure" can be influenced. Thus, approaches that influence any layer(s) of Fig. 2 are all candidate software restructuring approaches.

For example, reducing programmer turnover—a problem of the management layer of Fig. 2—may be viewed as a structure-preserving technique. Since programmers have invaluable insight into the behavior of software, high turnover of programmers can lead to information loss about the software. High turnover does not allow enough time to pass on this information before a person leaves. The result is a progressive net loss in software structure, which is typically manifested by software changes becoming harder to implement. The basic problem of preserving information among programmers may be ameliorated by decreasing programmer turnover.

Unfortunately, few authors—anywhere—define their notions of software structure in their papers or books. The reader should realize the "structure" in restructuring is often implicit, with the discussion context hinting at what structure is intended.

[1]For example, if a programmer returns to software he (or she) has written years earlier, it may initially appear strange and difficult to understand.

D. Software Restructuring is a Means to an End

Software restructuring should not be an end in itself. Restructuring should be related to locally defined goals, and achieving goals should be related to perceived software value. If restructuring cannot be justified in terms of higher goals, then these goals should be rethought, or more information collected, before the decision to restructure is made.

For example, if the problem is slow performance of user-requested changes in a software maintenance shop then maintenance management's higher goal of satisfactory software service is affected. The reason for the slow performance could be hard-to-modify software (in which case software restructuring may be advisable), or unrestrained user requests for change (in which case software restructuring may not be advisable), among other reasons. Further information is needed before one decides to restructure. But if restructuring is selected, it is because of the goal of satisfactory software service and not restructuring for its own sake.

A major goal of software restructuring is to preserve or increase software value. Software value can be measured externally or internally:

- External software value is the cost savings the software provides to the user community, relative to other, non-software means of satisfying user needs. For users, successful software maintenance is typified by few—preferably no—visible bugs in the software and rapid response to, and implementation of, user requests for system change. As noted earlier, systems undergoing maintenance often become progressively more difficult to change. If this progressive software ossification ever begins affecting the user's expectation of delivered software capability, the external value of the software may decrease.
- Internal software value involves at least three kinds of cost savings: 1) the maintenance cost savings that the software form provides relative to some other software form, 2) the cost savings incurred by reusing parts of the software in other systems, and 3) the cost savings due to an extended software lifetime (which delays the introduction of a replacement system). If software restructuring reduces maintenance costs, increases the software's potential for reuse in other software, and extends the software's lifetime, the internal value of the current software should increase!

There is little reason to believe that the same definition of value is used by all maintenance environments. Software value can be measured in other ways, such as savings in calendar time to implement software changes, increase in maintainer morale, increase in management respect for software maintenance, and so on.

But having *some* definition of software value can materially affect decisions to restructure. Any decision to use a software restructuring approach should try to quantify the added software value, as locally defined, which the approach will provide.

E. Software Restructuring and Software Maintenance

Software restructuring is most often applied to software undergoing maintenance, for this is where the lack of software structure becomes most evident (and expensive). Throughout this paper, software restructuring will be considered in the context of software maintenance.

However, software restructuring is also applicable during software development. This occurs especially when development activities are undertaken in an environment resembling "traditional" software maintenance (i.e., no quality assurance, deadline-driven, little if any testing, no code reviews, etc.). This can result in software becoming less and less structured.

To take a plausible example, in a large software project that has little reused code that takes several calendar years to build, and has little quality assurance, the potential for unstructured software toward project end can increase. The development activities prior to delivery could center on corrections to errors uncovered by tests and on enhancements requested by users due to changed needs since the system was contracted. If development is deadline-driven, the changes may be hurried and resemble patches, which can lead to worse software structure. In this situation, software restructuring might be advisable before the system is accepted for maintenance.

II. State of the Art in Software Restructuring Technology

As noted earlier, by changing the state of items at any of the levels in Fig. 2, "software structure" can be influenced. We can study restructuring approaches by the levels of Fig. 2 addressed by each approach.

This section briefly describes restructuring approaches. The approaches are divided into techniques, methodologies, and reverse engineering. Techniques (Section II-A) pertain to making changes at one (or a few) of the levels in Fig. 2. Methodologies (Section II-B) can pertain to all levels. Reverse engineering (Section II-C) pertains to repopulating, or bringing up-to-date, information in any of the levels.

Two caveats: this section is not intended to be exhaustive. It is intended to give the reader a feel for a wide range of restructuring approaches. Also, the classifications are not exclusive; some approaches could be placed in more than one section.

A. Techniques

A "technique" for this list is a restructuring approach that pertains to one (or a few) levels of Fig. 2. The restructuring techniques presented here correspond to the innermost five levels: code, documentation, programming environment/tools, software engineers, and management/policies.

1) Code: Code-oriented restructuring techniques pertain to modifying code directly, mostly without relying on information in the other levels. In the following sections they are divided into techniques based on coding style, packages and reusable code, control flow, and data.

a) Coding style: These restructuring approaches modify code to make it easier to understand, often without altering control structure or data structure.

- *Pretty printing and code formatting:* This restructuring technique textually restructures code by applying spacing between logical subparts, indentation of nested statements, one statement per line, and so on. This often can be done totally automatically and may be available as a compiler option for output listings.
- *Coding style standardization:* This restructuring technique textually restructures code by modifying code to conform to coding standard style guidelines. For example, a style manual may require that certain coding structures be avoided (e.g., ALTER statements in COBOL) and that standard keywords be used. This approach is mostly manual, but is often used with automated tools such as pretty printers and control flow restructurers. Some COBOL restructurers (see below) allow their output code to be tailored to better conform to local style guidelines.
- *Restructuring with a preprocessor* ([29] and others): The idea here is to replace sections of code with statements in another, presumably easier-to-understand, language. The new statements can be automatically expanded into statements of the original language. This approach has the advantage of allowing software to be selectively restructured, with statements recognized by the preprocessor, while leaving the remaining software unchanged. This is important because some people are leery of restructuring approaches that restructure more than is apparently necessary.

(b) Packages/reusable code: These approaches use software packages, or reusable code, to replace poorly structured software, or add to software.

- *Restructuring code for reusability [30]:* This technique takes existing code and puts it in a form for reuse. This often involves "cleaning up" software interfaces (e.g., removing superfluous parameters, reducing procedural side effects, and so on). Another aspect of this technique is changing a system to accept reusable code. Several examples of this follow.
- *Buy a package to replace an old system [16]:* Sometimes the best way to "restructure" a system is to replace it with a system with known structure. Restructuring is not the only way to solve problems with a system that is hard to understand, hard to change, and unreliable when changed. Since most packages do not make their source code available to users for modification, modifications to the packages must be done—at potentially substantial cost and on the vendor's timetable—through the package vendor.
- *Buy a software package to replace an old system; then extend the package [16]:* If it is risky or expensive to modify a system, a package may exist that both does the system's function and can be extended for new use. This is a straightforward approach, provided a suitable, adequately documented package can be found. Again, getting the package supplier's source code for the package can be a problem. Getting information on how the package works can be a bigger problem.
- *Buy a package to replace part of an old system [16]:* As the title implies, this approach replaces a software part that is particularly in need of restructuring. The approach depends on being able to find a suitable replacement package. The concerns raised above about vendor packages still apply.
- *System sandwich approach [16]:* This is an ingenious approach for retaining the benefits of code that is so badly structured it must be treated as a black box. The idea is to sandwich the old system between a new front-end interface (e.g., written in a fourth generation language) and a new back-end data base. The front-end interacts with the user. It also issues calls to the black box to compute information not currently available from the back-end data base. The black box system computes its outputs and directs it to the back-end database. The front-end and back-end can directly

communicate for report generation purposes. The old system is used primarily for its outputs to the back-end database.

c) Control flow: Much of the concern for software restructuring began with a concern for making a program's control flow easier to follow. This category contains algorithms and procedures for restructuring program control flow graphs, and tools for restructuring programs. The restructuring tools are mostly oriented to COBOL, for this is where most commercial interest in restructuring has been. The tools typically offer much more than just the ability to restructure code. For example, many tools offer style standardization, measurements of the code before and after restructuring (e.g., number of goto's), and/or automatically generated documentation (e.g., depicting control flow relationships among program elements) about the code. Since competition is motivating constant improvements the tool marketers should be consulted for latest information. No evaluation is implied by the presence or absence of a tool/approach in this list.

- *Early goto-less approach [11]:* This approach is famous for showing "goto's" are not theoretically necessary to create a computer program. The proof is by construction and contains a way to restructure software.
- *Giant case statement approach [7]:* This is another constructive way to remove goto's. The resulting program looks like a giant case statement.
- *Boolean flag approach [58]:* This is a procedure for creating a "structured" flowgraph by introducing Boolean variable(s).
- *Duplication of coding approach [58]:* This approach eliminates goto's to shared sections of code by duplicating the shared code and eliminating the sharing. This approach will not work for some looping programs.
- *Baker's graph-theoretic approach [8]:* This is the algorithm behind the tool "struct" for restructuring FORTRAN programs. Goto's are allowed on a limited basis.
- *Refined case statement approach [33]:* This approach introduces some procedures and heuristics to make the program resulting from the giant case statement approach easier to read and understand. This approach has some mathematical foundations in the work in [37].
- *RETROFIT (tm) [35]:* A tool for restructuring COBOL programs. Marketed by the Catalyst Group of Peat Marwick Main & Co., Chicago, Illinois.
- *SUPERSTRUCTURE (tm) [38]:* A tool for restructuring COBOL programs. Marketed by the Software Productivity Tools Division of Computer Data Systems, Inc., of Rockville, Maryland, through a recent acquisition of the previous marketer, Group Operations, Washington, D.C.
- *RECORDER (tm) [15]:* A tool for restructuring COBOL programs. Marketed by Language Technology, Salem, Massachusetts.
- *Cobol Structuring Facility (tm) [34]:* A tool for restructuring COBOL programs. Marketed by IBM, Bethesda, Maryland.
- *Delta/STRUCTURIZER (tm):* A tool for restructuring COBOL programs. Marketed by Delta Software Technologie AG, Switzerland.
- *Double conversion:* This approach takes a program in language *A*, uses an automated conversion tool to translate the program into language *B*, then uses another automated conversion tool to reconvert the program into language *A*. The result of this will usually be a program with different structure from the original.

d) Data: Historically, restructuring of software tends to connote restructuring of its control structure. But possibilities for restructuring data are important too. One example is putting the relations of a relational data base into third normal form. This has the advantage of reducing the need to propagate updates in a data base when data records are updated. Another example crops up in multidatabase environments. In such environments each database can have its own schema. Often, there are data items that mean approximately the same thing, but are named differently in each schema. To reduce the dependence of programs on the idiosyncrasies of data names with each database, the schemas might be restructured into a master schema that can hide lower level schema naming conventions. Programs can then be restructured to make queries using the master schema. This increases program understandability because programmers only have to know what data items in the master schema mean, rather than having to know the semantic nuances for each database.

2) Documentation: Documentation is often the first place programmers turn to before modifying code. Documentation helps the programmer understand code, plan and perform modifications, and perform testing. Unfortunately, documentation often goes out-of-date and then is never referred to. Missing or inconsistent documentation seems to be a constant complaint from maintenance programmers.

- *Upgrade documentation:* Examples of this are adding in-line code comments, making comments more accurate, expanding on cryptic commentary, and so on. Upgrading documentation and reverse engineering (discussed below) can overlap. The loose distinction we make is that upgrading documentation takes existing documentation and updates it, often without creating new forms of documentation. Reverse engineering may create new forms of documentation as well.
- *System modularization:* System modularization concerns how to decompose a proposed system into logically meaningful modules, or re-decompose an existing system into modules. System modularization currently requires much human judgment. Principles for manually performing modularization are available [41]–[43]. Some work on automating the modularization of systems has recently been reported [52].

3) Programming Environments/Tools: A programming environment provides a set of tools to assist the programmer in building, browsing and modifying software. Often the environment is used not just for programming, but for designing and creating requirements. In some environments the tools are integrated, where the environment provides support for sharing data among the tools.

- *Upgrade the programming environment:* Examples of this are adding windows to the programming interface, improving interaction of tools, replacing hard-to-use operating system command languages, and so on. These measures do not directly restructure target software, but they can increase the ability of a programmer to deal with software.
- *Programming environments/workstations ([9], [57] and others):* Programming environments offer more comprehensive support for programming needs than do tool col-

lections. For example, the programming environment may allow tools to be easily combined in a control procedure to create new tools. The programming workstation offers increased computing power to the programmer, along with new programming interfaces (e.g., windows).

- *Software metrics ([10], [23], [39], [47], and others):* The restructuring idea here is: 1) measure the software with a software metric (or a set of software metrics); 2) from the metric's value, answer the question, "Is the software property measured by the metric satisfactory?"; 3) if not, restructure the software and go to step 1); 4) if so, you are done.
- *Standards checkers and other aids:* These are tools that take a program and automatically report which software standards the program does and does not meet. Based on the reported violations, the code may be modified (restructured) to remove the violations.
- *Tool collections ([56] and others):* There is a growing number of tool collections that may be used to illuminate aspects of software structure. For example, MAP [56] will display the structure chart for a COBOL system, display a unit interface chart, highlight procedures in the structure chart that contain selected statements, display possible references of modifications to selected variables, and so on. MAP is now available as VIA/INSIGHT from VIASOFT, Phoenix, Arizona.
- *Program transformation systems [44]:* These systems involve automatic changes to software. The changes are accomplished with rules called transformations. This approach is related to the rule-based systems of artificial intelligence.
- *Fourth generation languages ([24] and others):* Though not generally thought of as restructuring tools, fourth generation languages offer significant benefits to the set of applications that may be rewritten in them. These benefits include ease of change, usability by end users, and quick development of small systems.

4) Software Engineers: The benefits of software structure must be perceived before they can be practically realized. Software problems (such as difficulty in modifying source code)—problems that normally suggest software modification-oriented restructuring solutions—may only be symptomatic of nonsoftware problems that impair or lose programmer's perceptions about software structure. Software restructuring is, in a very real sense, concerned with improving programmer's perceptions about software.

- *Train programmers:* Some common examples of this are the training courses in "structured programming" adopted by some companies, instruction on how to use existing tools to accomplish tasks, and advice from fellow programmers on how software works.
- *Hire new programmers, more experienced with the existing software application:* This is a most direct way to gain fresh insight into software structure. For example, if existing programmers are having difficulties modifying a windowing package to run on a new operating system, hiring a system programmer already experienced with the windowing package and with installing it might be advised. Even if the windowing package and the operating system are poorly documented so that they are difficult to understand, the new programmer's experience could compensate for these disadvantages. Of course, this may be only a short-term solution because the problem of the software being hard to understand by the general programming community may persist.
- *Reduce turnover:* When a person leaves an organization, a wealth of perceptions about software walks out the door. Steps to reduce turnover can help keep strong the group's consciousness about software structure.

5) Management and Policies: Management and policies can have a great impact on what people do with software, and therefore how software is perceived.

- *Programming standards and style guidelines ([28] and others):* This idea seems widely accepted, but surprisingly may not be put widely into practice [59].
- *Inspections and walkthroughs ([18], [20] and others):* This is one of the most effective practices for making software understandable and structure more recognizable.

B. Methodologies

A methodology is typically a set of steps for improving software at several of the levels shown in Fig. 2. A methodology helps direct the use of other, more specific, restructuring techniques.

- *System rejuvenation [16]:* System rejuvenation is defined as "using an existing system as the basis for a new strategic system." This is a methodology that involves cleaning up the existing system, making it more efficient (sometimes restructuring introduces a performance overhead), and putting the rejuvenated system into use.
- *Software Improvement Program [26], [27]:* This is an ambitious, management-intensive way to both restructure software and upgrade the software engineering practices of a maintenance environment. It consists of detailed technical guidance for planning the restructuring of software and improving the programming environment in which software is built.

Incremental restructuring [4]: Incremental restructuring is a restructuring approach without as much management overhead as the Software Improvement Program. The approach allows "structure" to be defined by users (rather than being built into the restructuring approach); restructuring is done in small, manageable parts. Also, a system can have the benefits of restructuring without having to be totally restructured. The approach is specifically designed to avoid introducing poor structure as a result of maintenance.

- *Software renewal [51]:* Software renewal is an approach not so much for modifying code as for upgrading system documentation, system specifications, and system tests.

C. Reverse Engineering

The reverse engineering techniques here emphasize recovering design information about existing code. Recreating complete documentation (including requirements documents) on existing code is often exorbitantly expensive and tends not to be done (e.g., see [45]).

- *Strategies for understanding software [19], [32]:* This approach suggests several heuristics for programmers to apply in trying to understand software that lacks up to date documentation. The steps in [19] are to learn the structure and organization of the program, determine what the program is doing, and document the program.
- *Design recovery [1], [2], [13], [40]:* These approaches suggest specific forms for recreated design information. In

some cases, automated tools are used to recreate the documentation.

- *Conversion [52]:* This approach creates information about a system as the system is being converted. The new information is used to assist the programmer in converting the system and in documenting the new system.

III. A Software Restructuring Case Study

Whenever one considers a restructuring technique, one is naturally curious about what results others have achieved. Supposing that the technique is a code restructuring tool, one can go to the tool vendor and inquire about the tool. Naturally one expects to get glowing praises and referrals to the most satisfied customers. The next source of information might be a users' group for the tool. Here one can expect more realistic appraisals of a tool's practical value. But the appraisals, though valuable, are very subjective. Other users gained experience with the tools in *their* environments with *their* programs; whether similar results will happen in one's own environment is unclear, given that environments and programs can differ dramatically.

Recently an attempt was made to compare the usefulness of maintaining restructured code with original code. Though methodological problems (discussed below) can be found in the study, the approach and results represent a noteworthy contribution to the measurement of restructuring efficacy.

The study, conducted by the Defense Logistics Agency[2] (DLA) of Columbus, Ohio, was an evaluation of a particular COBOL restructuring tool, RECODER (tm), from Language Technology[3], Salem, Massachusetts [21]. The goal of the study was to investigate claims made by the vendor about the restructuring abilities of the tool and to "evaluate the effects of the restructuring process on production code managed by DLA."

Fig. 3 presents the study plan. Six programs were selected by DLA to be restructured. Most of these programs were considered "unstructured." (See Table 1.) The programs were then restructured using RECODER. Programmers who were to work on the restructured code were given a training course taught by Language Technology. A set of measurements of recoded versus original programs was taken. The same set of changes were applied to both recoded and original programs. Different programmers were used to perform the changes. (No programmer was allowed to modify a program he or she wrote originally.) Measurements of the recoded and original software were taken again. (See Table 2.) Programmers were interviewed to get feedback about using the restructuring tool and about modifying the restructured programs.

The study resulted in several assessments. On the positive side, restructuring was generally beneficial: The recoded programs were easier to modify. (See Table 3.) The time to test the modified restructured programs was about the same as for the original programs. (See Table 4.) The recoded versions had better consistency in the coding style and documentation than the original versions. The recoded versions had significantly reduced violations in local standards and structure conventions. Programmers using the recoded versions "acquired increased knowledge levels and skills over those who worked with the unstructured versions." Programmers generally felt that the documentation generated by RECODER was helpful in understanding the recoded versions.

There were several negative observations: Several programs could not be restructured on the first attempt with RECODER.[4] The programmers expressed concern about the documentation style of programs produced by RECODER. The recoded programs had significant increases in resource utilization, such as compile time, load module size, and CPU resources. Programmers generally had to change paragraph names in the recoded versions to more meaningful names. Programs of poor quality to begin with were not considerably improved by restructuring. Some clean up of these programs prior to restructuring resulted in improvements in the code.

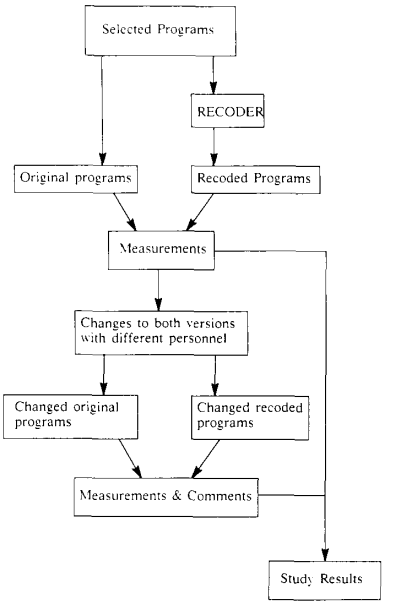

Fig. 3. Plan for studying the effectiveness of a restructuring tool [21].

Table 1 Programs Selected to Be Restructured

Program ID for Study	Program Name	Structured?*
PROG1	Policy Tables	No
PROG2	Prog. Data Ref. File Update	No
PROG3	Family & Cat. Chgs. to SCF	No
PROG4	Defense Inactive Item Prog.	No
PROG5	Stratification	No
PROG6	NON-NSN Demand Hist. Update	Yes

*Based on internal DLA assessment.

[2]Dr. M. Colter analyzed the study results and wrote the final report.

[3]Language Technology is to be commended for taking the risk of exposing their tool for such scrutiny.

[4]This ostensibly denied the vendor's claim that an ANSI standard COBOL could be restructured totally automatically. The problem lay with the standard, rather than the vendor, as the ANSI standard leaves room for interpretation and an instance of differing interpretations was uncovered here. Language Technology has since made changes so that the given programs can be restructured automatically.

Table 2 Sample Measurements Taken on Programs and Programmers

Measure	When Taken
Number of source lines	Before/after recoding
Number of statements	Before/after recoding
Number of standards violations	Before/after recoding
Compile service units, without optimization	Before/after recoding
Compile service units, with optimization	Before/after recoding
Load module size, without optimization	Before/after recoding
Load module size, with optimization	Before/after recoding
Execution (CPU) time	Before/after recoding
Revision time	After performing a modification on original and recoded programs
Test time	After performing a test on modified original and recoded programs

Table 3 Time to Modify Original and Recoded Programs*

Program ID for Study	Original Program	Recoded Program
PROG1	8	4
PROG2	1	1
PROG3	3	3
PROG4	1	1
PROG5	8	3
PROG6	2	1

*All times are in man-hours.

Table 4 Time to Test Original and Recoded Programs*

Program ID for Study	Original Program	Recoded Program
PROG1	1	1
PROG2	.5	1
PROG3	1	1
PROG4	2	.5
PROG5	1	1
PROG6	1	1

*All times are in man-hours.

As the report points out, several experimental methodology improvements are possible. Due to the study's design, it is not clear how much of these results are attributable to RECODER in particular, versus restructuring tools in general. No other restructuring tools were tried. Apparently the ability of the programmers was not made clear; the effect of programmer ability—a significant experimental variable to control—on the results is not clear. Not enough programs were selected to provide statistically significant results. Nevertheless, the study is a worthwhile start for the more rigorous investigation of the pros and cons of software restructuring.

IV. Software Restructuring Lessons Learned

A. Restructured Code Takes Some Getting Used To

When code is restructured, it is unfamiliar to programmers until they get used to its style. This depends on the style and supporting documentation created in the restructuring process. Often programmers react negatively at first to restructured code, but later like the code because its style is regular and predictable. Depending on the restructuring algorithm, the coding style of the restructured program may reflect modern programming practices—something which is a plus when hiring programmers recently graduated from curricula that stressed such program structuring principles. Older programmers not used to such principles can find the restructured code uncomfortable to deal with.

B. Restructuring Code for Large Systems is Often not Enough

For software many thousands of lines long, restructuring code to impart regularity and predictability in coding style is helpful, but modularization to further increase system comprehensibility is desirable. As noted earlier, modularization of code as part of its restructuring is still largely manual.

C. We Need Restructured Documentation Too

When code is automatically restructured, the in-line documentation can lose its relevance because the restructured code does not pose the same documentation referents. Since in poorly structured code the in-line documentation is often the final documentation (apart from the code itself) that the programmer depends on, the in-line documentation can be valuable to retain. Few restructuring tools attempt to modify inline code documentation so it can be transferred to restructured code.

D. The Programming Environment Affects How We Perceive Software Structure

Collections of tools, such as MAP mentioned above, allow different views of a program to be displayed one at a time. When a programmer moves from one view to the next, however, he or she is confronted with two problems: 1) retaining the information presented by each view, and 2) maintaining *relationships* between the views.

In terms of programmer understanding of software structure, the mental context switching needed as one moves from view to view can interfere with the gradual build-up of understanding about software. For example, a serial view of programming—e.g., view through an editor, view of compiler output, view of program output, view of debugger output—may have a high context switching overhead in the programmer's head. This could impair the programmer's ability to integrate the multiple views when solving restructuring problems.

Allowing programmers to see many software structure dimensions at once [54], with multiple windows or screens, may be more effective in restructuring than the typical serial views of software structure [50]. This ability ought to be "standard equipment" in programming environments.

E. Payoff of Restructuring can be Quantified and Predicted

Restructuring software alone does not assure a payoff. If the software is never modified, inspected, or reused again, then there is little opportunity to realize the potential gains of restructuring.

When selecting a restructuring approach for practical use, it is important to determine the approach's leverage—the ability of an approach to deliver effective results given the dollars invested. There are at least four factors that determine the leverage of a restructuring approach:

- The dollars invested to set up the approach
- The staff and facilities needed to support the approach
- The expected return of the approach
- The time frame for the return

The idea is, if the expected time frame for the return is satisfactory (e.g., does not exceed the expected remaining lifetime of the software), and the expected return (possibly including nonquantifiable benefits such as staff morale) significantly exceeds the return on the way maintenance is currently performed, then consider applying the restructuring approach. For some quantified models of this, see [12] and [4].

The time frame for the *quantifiable* return for most restructuring approaches is on the order of months to years. True, the effects of restructuring may appear immediately in the software itself, but the residual effects on maintenance economics can take much longer to appear, and even then must be considered with other factors before one can be sure improvements were due primarily to restructuring.

F. Systematically Decide How to Solve Problems With Restructuring

The software restructuring action plan in Table 5 outlines a way to select, apply, and evaluate a restructuring approach. The idea is first to discover the local, real maintenance problems. From these problems, one decides whether restructuring is the right approach at all. If so, then knowledge about the maintenance problems can be used to select from Section II a set of candidate restructuring approaches. According to the target level of software, local maintenance goals, and so on, candidate restructuring approaches may be selected. These approaches are then evaluated as to leverage, suitability in the local environment, and so on. Finally, a restructuring approach is used and evaluated.

Table 5 An Action Plan for Software Restructuring

1) Talk to maintainers about their perceptions of maintenance problems.
2) Identify current tasks where restructuring software might save staff time, reduce the maintenance budget, or achieve some other significant benefit.
3) Match an appropriate restructuring approach to the most pressing maintenance problems. A restructuring approach should be selected to have most impact on the tasks identified in step 2).
4) Do a feasibility analysis and a technology transfer analysis of the intended restructuring approach. A technology transfer analysis examines the social and psychological issues affecting acceptance and use of the restructuring approach in the workplace.
5) Select a restructuring technique, plan its use, and use it.
6) Monitor the restructuring effort, preferably by collecting data and applying measures of structure and of maintenance performance, and evaluate the results.

G. Take Steps to Preserve Software Structure After Restructuring

Software restructuring should be viewed as part of a more comprehensive solution to poor software structure. Once software is structured, presumably one would like it to stay structured with each software change. Here is where practices that foster good software structure come in—practices like defining and using software standards, giving programmers tools for checking conformance to software standards, performing code reviews, performing software tests with known degree of test coverage, quality assurance, and so on.

Quality assurance and restructuring are more related than one might think. Quality assurance applied from the beginning of maintenance can reduce the need for restructuring later on. If restructuring is required, quality assurance can help keep the software structured.

V. Future Work

Many software restructuring advances remain to be found. Here are some interesting topics for future research:

- *Deciding when and where to restructure:* Restructuring tools often go hand-in-hand with software metrics tools. The software metrics tools gather metrics that can help decide where and when to restructure. Despite research on systematic interpretation of metrics values and how to turn these interpretations into justifiable management actions (e.g., see [3]), there is still much work in guiding people in making restructuring decisions. What is needed are quantitative restructuring criteria based on validated metrics, along with practical results demonstrating efficacy.
- *Using restructuring for standardizing:* An important application of restructuring tools is to impose coding style standards. Tools are becoming increasingly parameterized to allow users more control in the style of programs produced. More work is needed to increase the tool's ability to accommodate many different standards.
- *Restructuring of documentation:* One problem with automated restructuring techniques that modify code is they don't restructure in-line documentation (i.e., rewrite program comments) along with the code. This means that manual labor to restructure documentation is nearly always needed after applying a restructuring approach. Automating the restructuring of documentation is important for making restructuring more cost-effective.
- *Modularization and design level information:* Design level information in the past was often not on-line, and hence not readily available for analysis and restructuring. Some CASE tools produce design information in the course of building software, and this information is available to restructuring tools. Relatively little work has been done in using this information for restructuring purposes.

Automatically restructuring systems by modularization is nearly uncharted territory. A Ph.D dissertation by Sobrinho [53] has started work in this area. More recently, [52] has discussed another approach for remodularization. It is unclear how effective these approaches are compared to manual remodularization. More work on new automatic (or

semi-automatic) modularization approaches is needed, along with studies demonstrating their effectiveness.

- *Graphical programming:* Graphical programming (e.g., [48]) offers a relatively new way to visualize, specify, and build software. Graphical programming may help alleviate some lower-level problems such as hard-to-understand control flow, and so change concern for software structure to other dimensions. Graphical programs could become a target output for reverse engineering tools.
- *Making available parts of restructuring tools:* Software restructuring tools are often highly proprietary. Users can adjust the restructuring process by using only the options made available in the tool's interface. Sometimes users may want to build their own special-purpose restructuring tools. Users might benefit from reusing parts used to build a proprietary tool. The idea here is that it may be profitable for both vendor and tool user to have "open architecture restructuring" tools. Users would have the option of using the complete package (i.e., the original tool), or using parts of the tool to accomplish restructuring. How to repackage tools into reusable parts useful to other restructuring tool builders is an open question.
- *Displaying software evolution:* One reason programmers have little clues about hard-to-understand software is that programmers have few tools for conveniently tracing how the software evolved to its current form. True, one can examine configuration management records, but a lot of detective work is often needed to recreate the programmer's mindset about the software that was changed. Evolution replay tools deserve consideration as ways to enhance understanding of software structure.

VI. Summary

Software restructuring is a tool for meeting maintenance goals and for increasing and preserving software structure. Software restructuring is part of a larger solution for maintaining the value of software as the software evolves.

Because software structure depends on programmer perceptions as well as the software state, software structure is dynamic. Steps must be taken to preserve structure in the minds of programmers, otherwise structure will be lost with programmer turnover.

There is a wide variety of restructuring approaches, ranging from approaches that do not modify software at all, but modify programmer perceptions of software, to those approaches that do modify software. The limit to what is a software restructuring approach is a gray area.

There is a lack of quantitative information about software restructuring. What we do know suggests restructuring leverage (the ability of a restructuring approach to "deliver" given the dollars invested) tends to come in the medium to long term (months to years). Even then, the effort to maintain software structure must be diligent, which may translate to higher quality assurance costs per maintenance change. Later, these costs are hoped to be justified through increased software flexibility (able to perform enhancements faster), reliability (fewer introduced bugs with each fix), lifetime (through extended usefulness to the enterprise), and reusability (due to the known software structure instilled in the software).

Software restructuring presents a very interesting research area. Besides the need for quantitative studies of restructuring effectiveness, work is needed in areas such as restructuring of documentation to correspond with restructured code, design restructuring, automatic application of software standards, automatic system modularization, and tools to reveal new aspects of software structure, such as graphical programming and software evolution animation.

Acknowledgments

The author wishes to thank Language Technology, Peat Marwick Main & Co., Group Operations (now part of Computer Data Systems, Inc.), IBM Corporation, Adpac, and Lexeme Corporation for information on the current status of their restructuring and restructuring-related tools. The author also thanks N. Schneidewind, D. Nettles, and B. Nejmeh, and an anonymous referee for their comments for improving the paper.

The following are Registered Trademarks: RECODER (of Language Technology, Inc.), RETROFIT (of Peat Marwick Main & Co.), Delta/STRUCTURIZER (of Delta Software Technologie AG), COBOL Structuring Facility (of IBM Corp.), VIA/INSIGHT (of Viasoft, Inc.), and SUPERSTRUCTURE (of Computer Data Systems, Inc.)

References

[1] P. Antonini, P. Benedusi, G. Cantone, and A. Cimitile "Maintenance and reverse engineering: low-level design documents production and improvement," in *Proc. Conference on Software Maintenance—1987*, IEEE Computer Society, 1987.

[2] G. Arango, I. Baxter, P. Freeman, and C. Pidgeon, "TMM: Software maintenance by transformation," *IEEE Software*, vol. 3, no. 3, May 1986.

[3] R. S. Arnold, "On the generation and use of quantitative criteria for assessing software maintenance quality," Ph.D. Dissertation, Computer Science Department, University of Maryland, College Park, 1983.

[4] ——,"Techniques and strategies for restructuring software," Notes for a software restructuring seminar conducted by R. S. Arnold, May 1985.

[5] ——,"An introduction to software restructuring," in *Tutorial on Software Restructuring*. Washington, DC: IEEE Computer Society, 1986.

[6] ——, *Tutorial on Software Restructuring*. New York, NY: IEEE Computer Society, 1986.

[7] E. Ashcroft and Z. Manna, "The translation of 'goto' programs in 'while' programs," in *Proceedings of the 1971 IFIP Congress*. Amsterdam, The Netherlands: North-Holland, 1971, pp. 250–260.

[8] B. Baker, "An algorithm for structuring flowgraphs," *J. ACM*, vol. 24, no. 1, pp. 98–120, Jan. 1977.

[9] D. R. Barstow, H. E. Shrobe, and E. Sandewall, *Interactive Programming Environments*. New York, NY: McGraw-Hill, 1984.

[10] V. Basili, *Tutorial on Models and Metrics for Software Management and Engineering*. Washington, DC: IEEE Computer Society, 1980.

[11] C. Bohm and G. Jacopini, "Flow diagrams, Turing machines, and languages with only two formation rules," *Commun. ACM*, vol. 9, no. 5, pp. 366–371, May 1966.

[12] L. Brice, "Existing computer applications. Maintain or redesign: How to decide?" in *Proc. of the Computer Measurement Group*, Dec. 1981. Reprinted in [6].

[13] R. N. Britcher and J. J. Craig, "Using modern design practices to upgrade aging software systems," *IEEE Software*, vol. 3, no. 3, May 1986.

[14] M. H. Brown and R. Sedgewick, "Techniques for algorithm animation," *IEEE Software*, vol. 2, no. 1, pp. 28–39, Jan. 1985.

[15] E. Bush, "The automatic restructuring of COBOL," in *Proceedings of the Conf. on Software Maintenance—1985* (Washington, DC), IEEE Computer Society, pp. 35–41, 1985.