Carnegie Mellon University
Software Engineering Institute

Understanding Program Dependencies

Norman Wilde
University of West Florida

August 1990

Preface

Capsule Description

A key to program understanding is unravelling the interrelationships of program components. This module discusses the different methods and tools that aid a programmer in answering the questions: "How does this system fit together?" and "If I change this component, what other components might be affected?"

Scope

In [Basili90], Victor Basili has said, "Most software systems are complex, and modification requires a deep understanding of the functional and non-functional requirements, the mapping of functions to system components, and the interaction of components." This module is about methods and tools for the *last* of these problems, that is, the understanding of interactions among program components.

Program components may be of many types, depending on the development methodology and the programming language used. However, this module will be restricted to those components that are found in code written in almost all conventional procedural programming languages: data items, data types, subprograms, and source files.

The module will present a classification of some of the main kinds of dependencies between these components, a discussion of the main methods for discovering the dependencies starting from the code, and a description of some of the ways that dependency information can be usefully presented to a programmer.

The general principles of tracing program dependencies are the same whether one is considering relationships in a large software system containing many (executable) programs, in a single program composed of many subroutines, or within a single subroutine. We will accordingly use the term "program dependencies" to refer to relationships at all three levels.

Since program dependencies can be complex, many software tools have been developed to help programmers comprehend them. Most of the methodologies that will be mentioned in this module have tool support, either a research prototype or more fully developed commercial tools. Although some tools are mentioned in the module, there is no implication that these are the only ones, or the best ones, available. Readers who are interested in finding tools for a specific problem or environment might consult some of the survey articles or tool directories that have been published. Some of these are referenced in the bibliography.

Philosophy

There are many reasons why analysis of program dependencies may be necessary. One good way to learn good coding practices is by reading existing code and observing how it is structured. Similarly, in designing a new system it may be useful to study the structure and relationships of existing systems. Program dependencies are also important in tasks such as parallelization, in which programmers must determine which calculations can take place simultaneously [Cowell90].

Undoubtedly the most common reason for trying to understand a program is that it needs to be changed. The maintenance phase of the software life cycle is almost universally found to be the most expensive in terms of programmer time; probably the largest fraction of that time is spent trying to analyze the program that needs to be maintained [Corbi89]. Component dependencies must be traced in order to find which components need to be changed and, later, to confirm that there are no undesired side effects of the change.

The best aid to program understanding is a human being who has experience with the system. People are much better than any existing tools for understanding the meanings of relationships and for knowing which ones are relevant to any particular problem. However, as systems grow in scale and lifetime, few organizations can afford to rely indefinitely on the availability of “the person who knows everything.”

When available human knowledge is not sufficient, there are tools and methodologies that can provide substantial assistance; the availability and use of these tools in programming shops is still much less than one would expect given the high costs of maintenance. This module provides background on the kinds of methodologies and tools that currently exist or that are becoming available in the hope of encouraging more widespread use.

Some reservations should be expressed about software analysis tools in general. First, while tools are likely to be a lot better than unaided hand analysis of code, there are still many cases in which tools either fail to capture some dependencies or else show relationships that do not really exist. Humans need to understand the strengths and weaknesses of the tools they use. Second, many organizations have found that simply purchasing tools will not automatically lead to increased productivity.

Leadership, training, and, most important, a well-defined software change process are needed to get any real benefit.

Acknowledgements

I would like to thank Lionel Deimel, Daniel Berry, Maribeth Carpenter, Mark Ardis, and Nancy Mead for many useful comments and suggestions during the development of this module, and particularly Gary Ford for his guidance and assistance in keeping the work focused. Sheila Rosenthal and Karola Fuchs in the SEI Library were extremely helpful in identifying and tracking down documents; without their efforts, the module could not have been prepared in twice the time.

Author's Address

Comments on this module are solicited, and may be sent to the SEI Software Engineering Curriculum Project or to the author:

Norman Wilde
Division of Computer Science
University of West Florida
Pensacola, Florida 32514-5750

Understanding Program Dependencies

Outline

Annotated Outline

1. Program Understanding

It is, of course, extremely dangerous to change a program that is not well understood. Modifications made in ignorance are very likely to cause subtle errors that will come back to haunt the programmer later or, at a minimum, will degrade the original program and make future maintenance more difficult.

On the other hand, *complete* understanding of a large system is an unrealistic goal. Rather, a maintainer must identify those program components that are important for a specific change and focus on understanding them well enough to safely make the modification. It is hard to define exactly how programmers go about achieving this level of understanding or even how they know when it has been achieved.

As systems age and grow, the difficulty of tracing dependencies almost certainly increases. For example, Belady and Lehman, in their classic study of the evolution of OS/360, show the growth in both the number of system modules and the number of modules that had to be handled in each new release [Belady76].

Software engineering theory might indicate that programmers should make use of system documentation for program understanding instead of relying only on program code. However, conversations with practicing maintainers indicate that they often have a deep distrust of all documentation that is separate from the code and frequently will not even pay much attention to program comments. When questioned about this seemingly negative attitude, they can usually relate experiences with misleading documentation and with the high cost of believing in it.

1.1. Theories of Program Understanding

Most descriptions of the program understanding process take either a top-down or bottom-up approach [Corbi89]. Brooks has recognized that the essence of high-level program understanding is not just in understanding the code, but also in the mappings between the program and its *problem domain*. To understand an inventory program, programmers may need to know something about warehouse operations, shipping orders, and the like; and they must be able to find how these things are reflected in the code. A programmer formulates hypotheses about the program in a top-down manner. Certain features in the code, such as variable names, are “beacons” that signal what is going on and allow hypotheses to be confirmed and refined [Brooks83].

Basili and Mills [Basili82] give a more formal method using a bottom-up approach, the goal being to derive from the code a specification that gives the function computed by the program. The method involves decomposing the program into smaller *prime programs* and determining the function computed by each prime. Then the functions are recombined to show the functions of larger program parts until the function of the whole program has been found.

Both of these methods indicate that understanding includes a process of gradually collecting information from the code of a program. This information is then used by the programmer to form higher level abstractions that explicate the program’s design; these can be used in planning program modifications (Figure 1).

To gather the necessary information, the programmer must usually trace code dependencies. For instance, starting from a question such as “Why is RECCTR being incremented here?” the maintainer will first look up the declaration of the variable, then find places where it is used, and then find places where its value is set. Perhaps in one of these investigations the programmer will find that RECCTR is being set equal to NUMREC, and this fact will set off another chain of inquiries. Eventually enough information

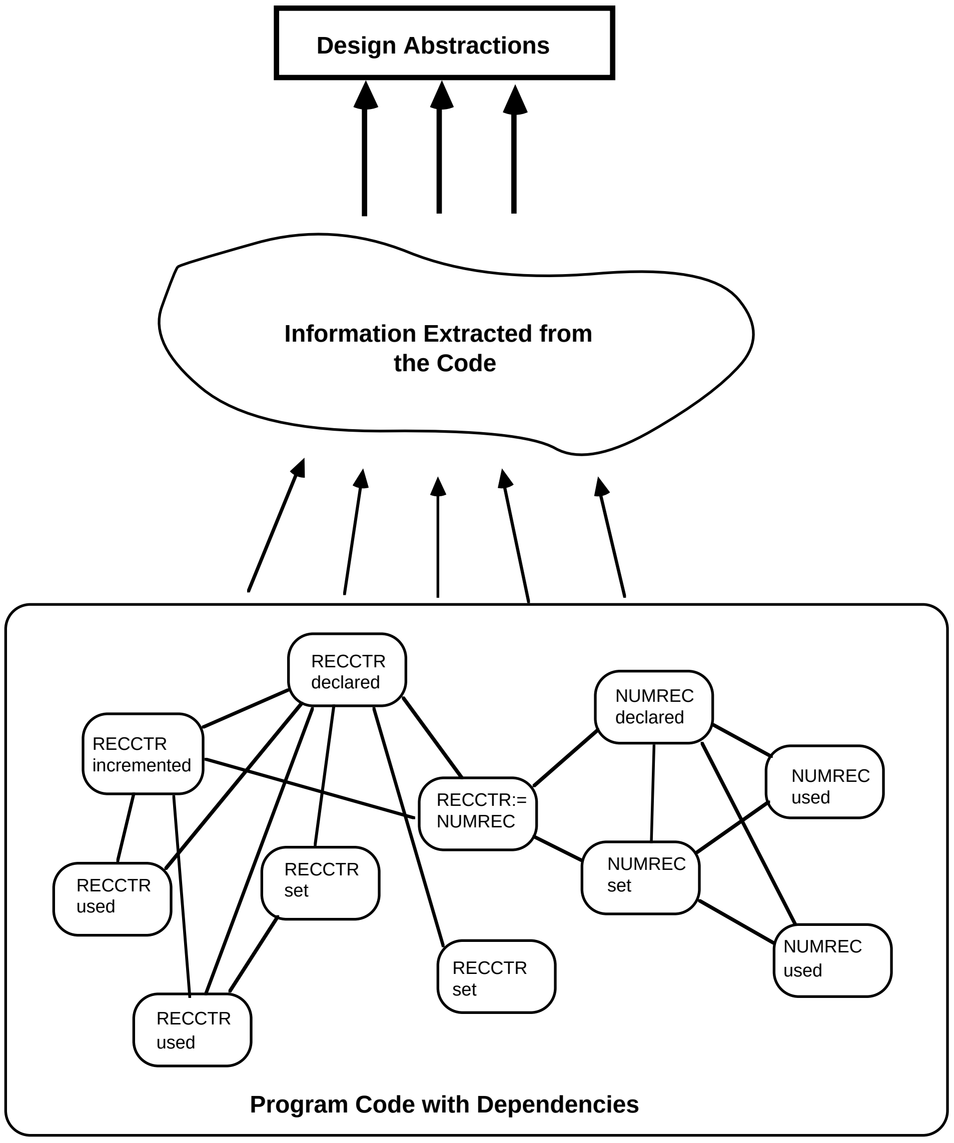

Figure 1. A typical process of program understanding

will be accumulated to allow the programmer to understand how RECCTR fits into the design abstractions of the overall program. While it is very difficult to describe all the mental processes that go on during the analysis, it is clear that tracing through the chains of dependencies is a large part of the work. Cleveland gives an interesting illustrative scenario of a programmer trying to track down a bug by searching through different views of the program code [Cleveland89].

1.2. Experiments and Practice in Program Understanding

Some experimental work on program comprehension has been reported in the literature. For example, Letovsky and Soloway have done experiments using the "thinking-aloud protocol" in which maintainers describe the mental processes they are using as they perform a task [Letovsky86]. The authors found that a major obstacle to program understanding is the presence of *delocalized* *plans,* in which a programmer uses a technique that is implemented in code distributed throughout the program instead of being located in one place.

Weiser hypothesizes that *program slices* (described in Section 3.4) may be used by programmers in code reading, especially in reading code to find a bug [Weiser82]. The work of Weiser and that of Letovsky and Soloway support the need for methods to pull together information about dependent program components that may not be contiguous in the code.

Fay and Holmes provide suggestions on how, in practice, one may attack the problem of modifying a program whose documentation is deficient [Fay85].

2. Classification of Program Dependencies

A software system may be structured in many different ways depending on the problem domain, design methodology, and implementation environment. But almost all software systems have components that are identifiable as data items, data types, subprograms, or source files. From the point of view of the maintainer, there is a *dependency* between two components if a change to one may have an impact that will require changes to the other.

2.1. Data Item Dependencies

Data items include variables, records, and structures. There is a dependency between two data items if the value held in the first affects or is used to compute the value held in the second. Most languages provide a rich collection of ways in which data dependencies may be established, ranging from assignment statements and computations to parameter matching and storage sharing.

2.2. Data Type Dependencies

Almost all languages have some built-in data types, and many allow the programmer to define additional ones, often by combining or modifying other types. Individual data items are usually declared to be of a specified data type. If the definition of a type is changed, attention must be paid to each such item and to each other type that has used it.

2.3. Subprogram Dependencies

We use the term *subprogram* in this module for those system components that actually carry out processing activities on the data. They may be *functions* in C, *functions* and *procedures* in Pascal, or even complete called programs in COBOL. Subprograms are related to each other by calling dependencies; if a subprogram is modified, the maintainer must be careful to check for effects on all the routines that call it. Subprograms also have dependencies with the data items that they use as input or that they compute.

2.4. Source File Dependencies

In many languages, source files may *copy* or *include* other source files. Physical copying is a frequent method of sharing system declarations such as data types or interface definitions between several software components. The real dependencies are not between the files, but rather between the components; the files are really just a packaging mechanism. However, in practice it is often more convenient to think in terms of dependencies between files because the files are the unit that is identifiable to the operating system or to the compiler.

A well-known problem is the difficulty of making sure that the executable code for a system is suitably updated when changes are made to just a few of the source files. Utilities such as *make* allow the programmer to specify source file dependencies that the utility will then use to rebuild the whole system as needed [Feldman79, Babich86].

2.5. Source Location Relationships

As has been mentioned, source files are a physical packaging mechanism for software components. Packaging does not normally create a *dependency* according to our definition since changing a component does not in any real sense require changes in the file. However, there is certainly a *relationship* that is of concern to a maintainer. Without extensive system experience or adequate tools, maintenance programmers are likely to spend a fairly large portion of their time trying to locate the files in which a given data type or subprogram is defined or used.

2.6. Additional Classes of Dependencies

Language designers have often provided other components for structuring software, such as the *objects* in C++ or the *packages* in Ada. The dependencies between these components can sometimes be handled by simple extrapolation from the methods discussed in this module, but such language features may also introduce complications that go beyond its scope. (See, for example, [Taenzer89] for a description of some of the complex dependencies in object-oriented languages.)

Certain software domains may contribute additional kinds of dependencies. Real-time software is a case in point since very troublesome effects may be introduced through timing relationships [Collofello85]. For example, a modification to one subprogram may cause it to consume more processor time, thus leading to a failure in an apparently independent task.

3. Finding Program Dependencies

3.1. Textual Search

Textual search is the simplest way to try to identify program dependencies. If one wants to find all the places where the data item INPUT_RECORD is changed, one solution is to search all the source code for the string “INPUT_RECORD”. Most text editors have commands for performing such searches on files. In many systems, utility programs exist for searching through many files at once, such as the *grep* utility on UNIX systems.

Textual search is useful but not very discriminating. Though we may only be interested in places where the data item is changed, we will also find places where it is used, written, mentioned in comments, etc. We may also find similarly named items (e.g., MASTER_INPUT_RECORD). In a large program written in a language that has some concept of identifier scope, there may even be many, totally different items with the same name.

3.2. Cross-Referencing

Cross-referencers seem to have appeared originally as a compiler option to aid in navigating around the code for a single executable module. Most compilers provide some kind of facility for listing where each identifier is declared and where it is used.

Some cross-referencers simply give the source file location of each reference and thus really provide location relationships, not dependencies. However, many are based on a more detailed parsing of the code and truly cross-reference the components, not their locations. Thus they can list, for example, all the subroutines that use global variable Z or all the variables declared to be of type X.

A problem with both textual search and many cross-referencing systems is that they depend on a specific name to represent the component being sought. In many contexts, however, there may be aliases for the component, created using storage sharing or pointers. A classic example comes from FORTRAN; if RECCTR appears in a COMMON area in one subprogram, then other subprograms that use the same area may access it under quite a different name. The user must check each occurrence of the reference to see if an alias is being established.

When a system becomes large, a simple listing of cross-references is likely to be unmanageable. Tools exist that scan all the system code–sometimes including components in different languages–and build cross-reference databases (e.g., [Chen86, Foster87]). The databases may be quite sophisticated so that they can distinguish different components having the same name and can identify how the component appears (for example, as a declaration, as input or output to a subroutine, mentioned in a comment, etc.) Listings may then be generated from the database, or online database queries may be made as necessary during a program understanding session.

3.3. Tracing Indirect Dependencies

One difficulty with both text search and cross-referencing systems is that they give only direct dependencies. However, maintainers often need to trace chains of dependencies to identify or understand a program function. For example, a typical dependency understanding task may require finding all places where TRACK_TABLE is searched, preparatory to making a change in its structure to improve efficiency. A query could show eight places where TRACK_TABLE is used. But in investigating the first of these, the programmer might discover that a pointer to the table is generated and passed to a subprogram as a parameter that is now named TAB_PTR. Inside the subprogram, TAB_PTR is passed to yet another subprogram where its name is again different, and so on. At this point, the maintainer would already be trying to track three levels of dependency without moving beyond the first reference to the table!

A solution to this problem is to consider the dependency relationships as a graph that can then be displayed or printed out in whole or in part [Wilde89]. This approach would seem to be generally applicable, but it has become common only for tools that show the calling hierarchy or *structure chart* of a program, which is traditionally represented as a tree or an indented listing [Kuhn87].

3.4. Data Flow Methods

Two methods of tracing program dependencies use detailed data flow analysis to define program subsets that are dependent. *Program slicing* answers questions such as, "What does the value of X at line 45 depend upon?" A program *slice* is an executable program found by removing from the original program those statements that have no effect on the specified value [Weiser81]. Slicing seems to be a very useful technique for detailed debugging, though it may be less useful in understanding broad program structure.

The ripple effect has been defined as "the phenomena by which changes to one program area have tendencies to be felt in other program areas" [Yau78]. *Ripple effect analysis* generally tries to answer the reverse of the program slicing question, "If the computation of Y at line 18 is changed, what other variables are affected and at which points in the program?" The answer is found by tracing data flow forward through the program from the point of the change [Yau78, Yau84].

Prototype research tools for both program slicing and ripple effect analysis have existed for some time, and there is some recent research on improving algorithms and extending the use of these techniques [Horwitz90, Yau88, Lyle88, Collofello88]. However, only a few commercial tools incorporating these methods seem to be on the market.

4. Some Practical Problems in Dependency Tracing

Most methods for finding program dependencies have practical problems that may limit the usefulness of a tool in a particular circumstance. There seems to be no systematic analysis in the literature of "what works when," so the following list of pitfalls is not exhaustive.

4.1. Preprocessors and Macro Substitution

Some programming languages, for example C and many assemblers, have a preprocessing phase in which macro substitutions are made to transform the input source code into a program that is ready for the compiler. Such substitutions complicate any analysis that goes much beyond textual search. Since the original source file may not, in fact, be a syntactically correct program, it is difficult to parse it reliably for purposes of data flow analysis. A solution often adopted is to analyze the code only after it has passed through the preprocessor. Unfortunately, some identifiers will be eliminated and others introduced by this process, so the output of the analysis may look unfamiliar to the user.

4.2. Arrays, Pointers, and Table-Driven Designs

Arrays and pointers may make it difficult for an analysis tool to determine exactly what is going on within a program. A reference to A[J] may be accessing any value within an array–or even outside it if bounds checking is not enforced. Pointer references may be similarly indeterminate. Many tools will try to state every dependency that *might* exist, thus overloading the user with excessive output that needs to be checked by hand. Good design practices, such as tables to define input and output records or arrays to define state machines, may create programs that can be analyzed only by very specialized tools.

4.3. Efficiency Considerations

Precise description and analysis of even a moderate-sized computer program may involve processing a surprising amount of information. In the current state of the art, large memory requirements and slow response time may pose significant barriers to the use of analysis tools on large-scale software systems.

4.4. Human Considerations

Maintainers, like most human beings, tend to be fairly conservative about their work. Many organizations have found that simply purchasing a tool is far from sufficient; the tool may be ignored by its intended users unless appropriate leadership, training, and motivation are present.

5. Visualizing Program Dependencies

Once dependencies have been found, by whatever means, the problem remains of presenting them to programmers in a way that improves their understanding of the program. The problem is not trivial because of the very large number of dependencies that may be present in a software system. For example, if a tool simply lays out the components as a graph on the screen with arcs representing dependencies, the result will usually be a spider web with little discernable structure. As another example, [Keuffel90] reports feeding a 138 line program into one tool, which responded with a 27,000 line report telling him everything the tool thought he needed to know about the program!

While no completely satisfactory solution to the visualization problem exists, there are many visualization methods that have been proposed and used with some success.

5.1. Annotated Listings

Despite the arrival of wide-screen workstations, program listings are still a very useful and widely used tool of the practical programmer. Some analysis tools enhance listings by *annotating* them with dependency information printed in the margins. Thus the procedure division of a Cobol program may be annotated with cross-references to the places variables are declared in the data division and the data division annotated with references to places variables are used in the procedure division. An interesting extension is the "book paradigm" suggested by Oman and Cook as a way of formatting code for easier understanding [Oman90a].

5.2. Constrained Queries

Some systems allow users to specify a query, stating exactly which kinds of dependencies they want to look at [Chen86, Wilde89, Rajlich88]. Basically this hands the problem back to the users. If they are experienced enough to be able to specify exactly what they want, then the tool will help them in program understanding tasks. If not, they may get so much output that the tool becomes useless.

5.3. Browsers

Browsers typically show the user a window into the code, along with a fairly small amount of dependency information indicating other program components that may be of interest. The user proceeds from one location to the next, acquiring information about the program that he or she can build into an overall understanding of its operation. For example, Cleveland has described one such browser, which allows the user to view the program from several different aspects [Cleveland89]. Related call graphs, control flow graphs, data usage, and data flow graphs can be displayed, along with the original code. (See [Oman90b] for several other examples of browsing tools.)

5.4. Clustering

One approach to the visualization problem for large systems is to use clustering methods to group components. Ideally, the clusters will turn out to be meaningful subsystems that enable programmers to get an overview of the software by looking at high-level subsystem dependencies instead of low-level component dependencies. Hutchens and Basili present an approach using hierarchical clustering methods based on the data bindings that link subprograms [Hutchens85]. Schwanke and Platoff describe an alternative system that groups components having similar intercomponent dependencies [Schwanke89].

5.5. The Grid Mechanism

Ossher has defined the *grid mechanism*, a graphical notation for describing the structure of large software systems and the interactions of their parts. The grid specifies which parts are allowed to interact with which others, so that designers can specify which units are intended to be "hidden" in the system. They can also specify exceptions to the overall structure, such as those that occur when a program is tuned for efficiency. While the grid cannot, at present, be generated automatically from the code, a grid description created by hand can be checked against the code for consistency [Ossher89].

6. Commercial Tools for Maintenance

Many commercial tools have been developed as an aid to software maintenance. Trade publications often contain reviews of these tools (see, for example, [Keuffel90]), and several surveys are available as an aid to finding a tool for a particular problem and environment [Federal86, Holbrook87, Zvegintzov89].

7. Reverse Engineering of Software

An active and exciting current research area is the development of methodologies for the *reverse engineering* of software. Reverse engineering is the process of analyzing a subject system to:

- identify the system's components and their interrelationships.
- create representations of the system in another form or at a higher level of abstraction.

Reverse engineering usually includes some kind of semiautomatic attempt to help the programmer go directly from the code level of Figure 1 (page 3) to the abstraction level. Most reverse engineering tools are still in the research stage, however, so they will not be discussed in detail in this module. The reader may want to consult the January 1990 special issue of *IEEE Software* for several papers showing the current state of this research.

Teaching Considerations

Prerequisites

Program dependencies may be discussed in any classroom setting in which the students have a basic background in writing and debugging non-trivial programs; this level of familiarity would normally be found, for instance, in students who have had an introductory programming course plus a data structures course. Of course, the material will be more meaningful to students who have some experience with systems of greater size and complexity, either in industry or in a course that has a large programming project.

Recommended Module Uses

In a Software Engineering Lecture Course

Objectives and Content: In a typical software engineering course, sometimes as little as one class hour can be devoted to maintenance topics. In this case the instructor can hope to do little more than make students aware of the importance of the problems associated with program understanding and maintenance. The paper by Corbi provides good background for preparing a lecture on this topic [Corbi89].

However, program understanding, its importance in maintenance, and relevant methodologies would seem to merit a somewhat deeper study even in such a course. If two to three class hours are available, the following objectives may be reasonable.

At the end of the unit a student should:

- Know the importance of maintenance within the life cycle and the importance of program understanding in maintenance.
- Understand and be able to explain the problem of comprehending a program's design when important pieces of the puzzle are not contiguous in the code (i.e., the "delocalized plans" problem).
- Know of the existence of tools for textual search and cross-referencing of code.

If more time is available, the content could be expanded in one of two ways. First, more advanced methods could be discussed, such as program slicing and ripple analysis. Alternatively, a small code-reading exercise could be given, requiring students to use available tools to answer questions about some moderately-sized program. For example, students might be given a program of about 500-1000 lines and asked to identify the purpose of a particular variable or locate where a particular task is being carried out. Deimel and Makoid give examples of such exercises [Deimel85].

Resources: As an overview of program understanding, one good starting point is the paper by Corbi [Corbi89], which may be used by the instructor in preparing lectures or assigned to the students for reading. A second good reference is the Letovsky and Soloway paper [Letovsky86], which clearly explains the concept of a "delocalized plan" and the problems such plans cause.

For students who have little practical experience, it may be useful to provide some exposure to the realities of a maintenance task. The paper by Fay and Holmes [Fay85] could provide useful reading.

An exercise that may be interesting is to ask students to review one or more tools, based on papers such as the ones in this module's bibliography or on product literature from commercial vendors. In one limited experience, I have found that students who have experienced the drudgery of maintenance can easily grasp the benefits of tool support; for an inexperienced class, however, this sort of exercise may be less useful.

In a Maintenance-Oriented Project Course

Objectives and Content: Maintenance-oriented projects are increasingly replacing or supplementing traditional development-oriented projects in computer science and software engineering education [Tomayko89, Cornelius89, Morris88]. Such project courses attempt to redress the imbalance in favor of development in the rest of the curriculum and expose students to problems more closely approximating real world experience. Within the context of these courses, some time could be devoted to lectures on the theory and practice of program understanding, and tools could be provided for student use. Morris provided his students with an initial small bug-finding exercise to familiarize them with the available software tools and found that they thereafter "… continued to use them throughout the course without further prompting."

The objectives in such a course would include those of the lecture course described in the previous section, but also the student should be able to:

- Analyze an existing system written by someone else, in order to identify a bug or plan an enhancement.
- Apply at least one or two tools selected by the instructor as support for this program analysis task.

Exercises may start at a simple level. For example, students could be asked to locate where a particular variable is set or used. Further exercises could then go on to more complex problems in which students are required to identify those components in which some particular function is being carried out, and finally to problems involving actual bug fixes and enhancements to the program.

Resources: The papers mentioned at the beginning of this section are a useful source of ideas on the organization of a maintenance-oriented project course. They provide useful advice on the selection of a software artifact to study, a possible sequence of exercises, etc. In addition, [Engle89] provides an Ada program with associated documentation that can be used as raw material for such a course. The notes provided with the program include a series of suggested exercises ranging from developing documentation and management plans through code reviews and actual system modifications.

Project Suggestions

The specification and development of a program understanding tool can be a challenging and interesting project for a student team in a project course. The possibilities are extensive and can range from quite easy projects to systems of considerable difficulty.

Cross-referencing tools are fairly simple to build and can be appropriate for lower level courses, though the interface design may still prove challenging. Instructors may want to consider going beyond the more common programming languages; a recent student project supervised by the author specified a dependency analysis tool for spreadsheets.

For advanced courses, the design of a program slicer or ripple analyzer will require some library research on algorithms and a good deal of careful analysis. Some of the problems in handling interprocedural data flow, pointers, and arrays are far from completely resolved. Students should be thoroughly conversant with the language being analyzed; it may be prudent to start with a language subset rather than attempting to handle all constructs of a major programming language.

Bibliography

Ambras88

Ambras, James P., Berlin, Lucy M., Chiarelli, Mark L., Foster, Alan L., O'Day, Vicki, and Splitter, Randolph N. "MicroScope: An Integrated Program Analysis Toolset." *Hewlett-Packard Journal* (Aug. 1988), 71-83.

This article describes a tool that analyzes Lisp programs. "The current system includes a static component that analyzes the cross-reference structure of a program and a dynamic component that lets users monitor the run-time behavior of the program."

Babich86

Babich, Wayne A. *Software Configuration Management: Coordination for Team Productivity* Reading, Mass.: Addison-Wesley, 1986.

Chapter 7 provides a brief but clear overview of the *make* utility and of source file dependencies. (See [Feldman79] for a more detailed treatment)

Basili82

Basili, Victor R., and Mills, Harlan D. "Understanding and Documenting Programs." *IEEE Transactions on Software Engineering SE-8,* 3 (May 1982), 270-283.

Abstract: *This paper reports on an experiment in trying to understand an unfamiliar program of some complexity and to record the authors' understanding of it. The goal was to simulate a practicing programmer in a program maintenance environment using the techniques of program design adapted to program understanding and documentation; that is, given a program, a specification and correctness proof were developed for the program. The approach points out the value of correctness proof ideas in guiding the discovery process. Toward this end, a variety of techniques were used: direct cognition for smaller parts, discovering and verifying loop invariants for larger program parts, and functions determined by additional analysis for larger program parts. An indeterminate bounded variable was introduced into the program documentation to summarize the effect of several program variables and simplify the proof of correctness.*

Basili90

Basili, Victor. "Viewing Maintenance as Reuse-Oriented Software Development." *IEEE Software* 7, 1 (Jan. 1990), 19-25.

Belady76

Belady, L., and Lehman, M. “A Model of Large Program Development.” *IBM Systems Journal 3* (1976), 225-252.

Abstract: *Discussed are observations made on the development of OS/360 and its subsequent enhancements and releases. Some modeling approaches to organizing these observations are also presented.*

This is the classic paper on the ways systems grow and evolve. The paper proposes the following laws of program evolution, which are relevant background for software maintenance and program understanding:

> “1. Law of continuing change: A system that is used undergoes continuing change until it is judged more cost effective to freeze and recreate it.
>
> “2. Law of increasing entropy: The entropy of a system (its unstructuredness) increases with time, unless specific work is executed to maintain or reduce it.”

Brooks83

Brooks, Ruven. “Towards a Theory of the Comprehension of Computer Programs.” *Int. J. Man-Machine Studies 18* (1983), 543-554.

Abstract: *A sufficiency theory is presented of the process by which a computer programmer attempts to comprehend a program. The theory is intended to explain four sources of variation in behavior on this task: the kind of computation the program performs, the intrinsic properties of the program text, such as language and documentation, the reason for which the documentation is needed, and the differences among the individuals performing the task. The starting point for the theory is an analysis of the structure of the knowledge required when a program is comprehended which views the knowledge as being organized into distinct domains which bridge between the original problem and the final problem. The program comprehension process is one of reconstructing knowledge about these domains and the relationship among them. This reconstruction process is theorized to be a top-down, hypothesis driven one in which an initially vague and general hypothesis is refined and elaborated based on information extracted from the program text and other documentation.*

This paper, together perhaps with [Basili82], would be the best source of a theoretical framework for teaching about program understanding.

Chen86

Chen, Yih-Farn, and Ramamoorthy, C. V. *The C Information Abstractor*. Report No. UCB/CSD 86/300, Computer Science Division (EECS), University of California, Berkeley, Calif., June 1986.

Abstract: *Program understanding is one of the most time-consuming processes in software maintenance. This is partially due to the human inability to memorize complex interrelations among the software entities of a large software system. The situation worsens when the programs are not written by the software maintainers and little documentation is available. The basic idea in* information abstraction *is to extract relational information among the software entities of programs, store the information in a database, and make it available to users in a form that can be easily understood. We have imple-*

mented an Information Abstractor to extract relational information from C programs and store the information into a program database. High level access utilities are provided so that program maintainers or developers can easily retrieve the information they need for understanding the software. Besides program understanding, we found that the availability of the program database can also promote the research in four software engineering areas: multiple software views, software reusability, software metrics, and software restructuring.

A good example of a cross-reference database tool.

Chikofsky90

Chikofsky, Elliot, and Cross, James, II. "Reverse Engineering and Design Recovery: A Taxonomy." *IEEE Software 7*, 1 (Jan. 1990), 13-17.

Cleveland89

Cleveland, L. "A Program Understanding Support Environment." *IBM Systems Journal 28,* 2 (1989), 324-344.

Abstract: *Software maintenance represents the largest cost element in the life of a software system, and the process of understanding the software system utilizes 50 percent of the time spent on software maintenance. Thus there is a need for tools to aid the program understanding task. The tool described in this paper–Program UNderstanding Support environment (PUNS)–provides the needed environment. Here the program understanding task is supported with multiple views of the program and a simple strategy for moving between views and exploring a particular view in depth. PUNS consists of a repository component that loads and manages a repository of information about the program to be understood and a user interface component that presents the information in the repository, utilizing graphics to emphasize the relationships and allowing the user to move among the pieces of information quickly and easily.*

This is an interesting paper that shows clearly how sophisticated browsers can be used to support maintenance tasks. It provides a good scenario of browser use.

Collofello85

Collofello, James, and McBride, David. "Maintenance Performance Ripple Effect Analysis for Real-time Ada Programs." *Proc. Compsac 85,* Chicago, IEEE Computer Society, October 1985, 17-23.

Abstract: *The complexity and expense of performing software maintenance on large-scale programs is well known. Since large-scale programs often possess both a set of functional and performance requirements, it is important for maintenance personnel to consider the ramifications of a proposed modification from both a functional and a performance perspective.*

This paper describes the possible ripple effect of program modifications during the maintenance phase on the performance of a program and presents a technique for the analysis of this performance ripple effect in large-scale Ada programs. The significance of this type of maintenance technique is its contribution to an engineering approach to large-scale software maintenance. By predicting the repercussions generated by software modifications, it can aid

maintenance personnel in their selection of modification alternatives. It can also help in the retesting phase to determine whether any performance requirements have been violated by the maintenance activity.

Collofello88

Collofello, James, and Orn, Mikael. "A Practical Software Maintenance Environment." *Proc. Conference on Software Maintenance - 1988,* Phoenix, Arizona, IEEE Computer Society, October 1988, 45-51.

Abstract: *This paper provides an update of a research project at Arizona State University whose objective is the development of a practical software maintenance environment. The existing functional capabilities of the environment are described as well as research currently in progress.*

The environment described in this paper works on Pascal code and includes tools for generating structure charts, displaying import and export data from each module, accessing text documentation and module code, and performing ripple effect analysis. The ripple analysis tool is particularly interesting since it makes use of semantic information to localize ripple effects more precisely.

Corbi89

Corbi, T. A. "Program Understanding: Challenge for the 1990s." *IBM Systems Journal 28,* 2 (1989), 294-306.

Abstract: *In the Program Understanding Project at IBM's Research Division, work began in late 1986 on tools which could help programmers in two key areas: static analysis (reading the code) and dynamic analysis (running the code). The work is reported in the companion papers by Cleveland and by Pazel in this issue. The history and background which motivated and which led to the start of this research on tools to assist programmers in understanding existing program code is reported here.*

Corbi provides a very good overview of program understanding and its importance in maintaining old systems. It is good background for the instructor or may be assigned for student reading.

Cornelius89

Cornelius, B. J., Munro, M., Robson, D. J. "An Approach to Software Maintenance Education." *Software Engineering Journal 4,* 4 (July 1989), 233-236

Abstract: *The majority of courses in software engineering concentrate on educating students in the methods applicable to the development stage of the software lifecycle. Software maintenance is recognized as the most expensive phase of the software lifecycle, and yet it receives very little attention from those involved in software engineering education. This paper describes a novel approach to software maintenance education which is being undertaken at the University of Durham.*

This is a good paper for instructors to read in preparing a maintenance-based project for students.

Cowell90

Cowell, W. R., and Thompson, C. P. "Tools to aid in discovering parallelism and localizing arithmetic in Fortran programs." *Software Practice and Experience 20*, 1 (Jan. 1990), 25-47.

Abstract: *We describe a collection of software tools that analyze and transform Fortran programs. The analysis tools detect parallelism in blocks of code and are primarily intended to aid in adapting existing programs to execute on multiprocessors. The transformation tools are aimed at eliminating data dependencies, thereby introducing parallelism, and at localizing arithmetic in registers, of primary interest in adapting programs to execute on machines that can be memory bound (common for machines with vector architecture).*

Deimel85

Deimel, Lionel, Jr., and Makoid, Lois. "Developing Program Reading Comprehension Tests for the Computer Science Classroom." *Computers in Education*, Duncan and Harris (eds.), Elsevier Science Publishers, 1985, 535-540

Abstract: *A methodology for constructing program reading comprehension tests is discussed and illustrated. Emphasis is on multiple-choice tests used with realistic reading passages. Item writing employs a classification of question types developed by the authors and a program comprehension model developed by Ruven Brooks is recommended.*

This paper provides guidance on developing simple code-reading exercises and tests. It includes several examples.

Engle89

Engle, C. B., Jr., Ford, G., and Korson, T. *Software Maintenance Exercises for a Software Engineering Project Course*. Educational Materials CMU/SEI-89-EM-1, Software Engineering Institute, Carnegie Mellon University, Pittsburgh, Pa., Feb. 1989.

Abstract: *Software maintenance is an important task in the software industry and thus an important part of the education of a software engineer. It has been neglected in education partly because of the difficulty of having to produce and then maintain a software system within a semester course. This report provides an operational software system of 10,000 lines of Ada and several exercises based on that system. Concepts such as configuration management, regression testing, code reviews, and stepwise abstraction can be taught with these exercises.*

This is the first SEI package of educational materials. The package includes documentation and source code for an Ada system. The exercises provide a good basis for organizing a maintenance-oriented student project.

Fay85

Fay, Sandra, and Holmes, Denise. "Help! I Have to Update an Undocumented Program." *Proc. Conference on Software Maintenance - 1985,* Washington, D. C., IEEE Computer Society, November 1985, 194-202.

Abstract: This paper discusses a method for documenting and maintaining an undocumented program. The paper provides guidance to junior personnel and management of areas that can alleviate the situation.

The paper specifically addresses: First impressions; Resources, who and what; Approaches, Schedule assessment.

This paper is directed to those people in industry who are faced with documenting an undocumented program. However, it is also written with the hope that this will give the person supervising the maintainer a clearer view of the help which can be given by providing the resources and time necessary to maintain a program in the proper manner.

An excellent paper to give students; it obviously represents considerable experience in handling both real world code and real world managers!

Federal86

Software Aids and Tool Survey. Federal Software Management Support Center, Office of Software Development and Information Technology, Report OIT/FSMC-86/002, 1986. Available from the U.S. Government Printing Office, as GPO 022-002-00106-2.

A very extensive classification and survey of tools, but with no evaluation.

Feldman79

Feldman, Stuart I. "Make - A Program for Maintaining Computer Programs." *Software Practice and Experience 9,* 4 (Apr. 1979), 255-265.

***Summary:** Good programmers break their projects into a number of pieces, each to be processed or compiled by a different chain of programs. After a set of changes is made, the series of actions that must be taken can be quite complex, and costly errors are frequently made. This paper describes a program that can keep track of the relationships between parts of a program and issue the commands needed to make the parts consistent after changes are made.* Make *has been in use on UNIX systems since 1975. The underlying idea is quite simple and can be adapted to many other environments.*

This paper describes source file dependencies and the *make* utility for handling selective system rebuilds. It would be appropriate as a starting point for students who will be using such a tool, though they may also need to consult local manuals since some variants of *make* now exist.

Foster87

Foster, John, and Munro, Malcolm. "A Documentation Method Based on Cross-Referencing." *Proc. Conference on Software Maintenance - 1987,* Austin, Texas, IEEE Computer Society, September 1987, 181-185.

***Abstract:** Much of the work of the maintenance programmer involves the investigation of program operation by the examination of the source code itself. The purpose of such examination is to discover information about the program and its components: the information gained is potentially valuable to future maintenance work as well as to the immediate task in hand. Unfortunately,*

the urgency of the typical maintenance task is such that the information will rarely be recorded in an adequate form, or even at all.

The paper describes a method which can be used to document the results of maintenance investigations and which is currently under implementation as a toolset. The core of the toolset (and the method) is an advanced cross-reference and indexing tool for source and documentation. The method allows a listing of program components and functional attributes to be constructed incrementally, under the control of an interactive interface. Despite the interactive nature of the main interface, the method is suitable for use under a strict Quality Assurance environment, and the manner of achieving this is described.

An example of a tool (developed by British Telecom) suitable for use on large-scale, multi-language systems. This is a good paper for students who wish to get an idea of the sorts of tools that are feasible.

Holbrook87

Holbrook, H. B., and Thebaut, S. M. *A Survey of Software Maintenance Tools That Enhance Program Understanding.* Report SERC-TR-9-F, Software Engineering Research Center, Computer and Information Sciences Department, University of Florida, Gainesville, Fla., 1987.

Abstract: *This report summarizes the results of a recent survey of commercially available software tools which purport to aid in the task of program understanding. The effort was undertaken in connection with the SERC "Maintenance Assistant" research project at the University of Florida during the summer of 1987, and resulted in the identification of 116 tools. Most of the tools identified provide insight into the program structures and operations considered important for program comprehension.*

Another listing and classification of commercial tools.

Horwitz90

Horwitz, Susan, Reps, Thomas, Binkley, David. "Interprocedural Slicing Using Dependence Graphs." *ACM Transactions on Programming Languages and Systems 12*, 1 (Jan. 1990), 26-60.

Abstract: *The notion of a* program slice, *originally introduced by Mark Weiser, is useful in program debugging, automatic parallelization, and program integration. A slice of a program is taken with respect to a program point* p *and a variable* x*; the slice consists of all statements of the program that might affect the value of* x *at point* p*. The paper concerns the problem of interprocedural slicing-generating a slice of an entire program, where the slice crosses the boundaries of procedure calls. To solve this problem, we introduce a new kind of graph to represent programs, called a* system dependence graph*, which extends previous dependence representations to incorporate collections of procedures (with procedure calls) rather than just monolithic programs. Our main result is an algorithm for interprocedural slicing that uses the new representation. (It should be noted that our work concerns a somewhat restricted kind of slice; rather than permitting a program to be sliced with respect to program point* p *and an* arbitrary *variable, a slice must be taken with respect to a variable that is* defined *or* used *at* p*.)*

The chief difficulty in interprocedural slicing is correctly accounting for the calling context of a called procedure. To handle this problem, system dependence graphs include some data dependence edges that represent transitive *dependences due to the effects of procedure calls, in addition to the conventional direct-dependence edges. These edges are constructed with the aid of an auxiliary structure that represents calling and parameter-linkage relationships. This structure takes the form of an attribute grammar. The step of computing the required transitive-dependence edges is reduced to the construction of the subordinate characteristic graphs for the grammar's nonterminals.*

This paper would be appropriate for a group of advanced students interested in building their own slicer.

Hutchens85

Hutchens, David, and Basili, Victor. "System Structure Analysis: Clustering with Data Bindings." *IEEE Transactions on Software Engineering SE-11,* 8 (Aug. 1985), 749-757.

Abstract: *This paper examines the use of cluster analysis as a tool for system modularization. Several clustering techniques are discussed and used on two medium-sized systems and a group of small projects. The small projects are presented because they provide examples (that will fit into a paper) of certain types of phenomena. Data bindings between the routines of the system provide the basis for the bindings. It appears that the clustering of data bindings provides a meaningful view of system modularization.*

Keuffel90

Keuffel, Warren. "Making Sense of it All: Groupware for Re-Engineering." *Computer Language 7,* 6 (June 1990), 93-101.

Abstract: *This review covers a number of products that share a common focus: documenting existing code so that it can be understood and maintained by any member of a group. Some of these products are unique; some offer overlapping functions. All are worth considering.*

An example of a product review from the trade press, this article focuses mainly on tools for PC code.

Kuhn87

Kuhn, D. Richard. "A Source Code Analyzer for Maintenance." *Proc. Conference on Software Maintenance - 1987*, Austin, Texas, IEEE Computer Society, September 1987, 176-180.

Abstract: *This paper describes a tool that reads all C source files in a directory and produces information useful for program maintenance. The tool generates a call tree, a call matrix, and the transitive closure of the matrix, which shows indirect relationships between routines. It computes some measures that may help estimate the complexity of the program being maintained, and also identifies subsystems (possibly nested) within the program. The paper describes the information provided and shows how it saves time in understanding the program to be modified, estimating the complexity of the change, and performing regression testing on the modified program. The tool is in the public domain and will be available through the National Technical Information Service (NTIS).*

An example of a tool that produces calling hierarchy charts. The tool is available from the paper's author and may be appropriate for classroom use.

Letovsky86

Letovsky, Stanley, and Soloway, Elliot "Delocalized Plans and Program Comprehension." *IEEE Software 3*, 3 (May 1986), 41-49.

A very interesting description of the problems programmers encounter in understanding code. Several very illustrative examples are based on experience observing programmers as they work. There is also a discussion of the implications for better program documentation. This is a good paper to give students to get them thinking about how to read code and how to write code that will be readable.

Lyle88

Lyle, J. R., and Gallagher, K. B. "Using Program Decomposition to Guide Modifications." *Proc. Conference on Software Maintenance - 1988*, Phoenix Arizona, IEEE Computer Society, October 1988, 265-269.

Abstract: *We use data flow techniques to form a notion of direct sum decomposition for programs. The decomposition yields a method and guidelines for "software surgeons" (maintainers) to use so that changes can be assured to be completely contained in the modules under consideration and that there are no undetected "linkages" between the modified and the unmodified code. Thus, the impact of small changes can be gauged. The decomposition can also be used to limit the amount of testing required to assure that the change is correct; in fact, under suitable conditions modification testing will be required only for the changed code. Moreover, if these hypotheses are violated, the modifier can be virtually assured that proposed changes will have a wider impact than that which is contemplated.*

Morris88

Morris, Robert A. "An Unorthodox Approach to Undergraduate Software Engineering Education." *Computing Systems 1,* 4 (Fall 1988) 405-419

Abstract: *Software engineering principles can be taught to inexperienced undergraduates by substituting code reading, maintenance, and enhancement for the more usual beginning-to-end team project. The study of mail reading systems of intermediate size proves a suitable environment for the study of complex systems.*

This is another useful paper for the instructor who is planning a maintenance-oriented student project.

Oman90a

Oman, Paul, and Cook, Curtis. "The Book Paradigm for Improved Maintenance." *IEEE Software 7,* 1 (Jan. 1990), 39-45

This paper describes the "book paradigm," a way of formatting the listing of a program like a book in order to facilitate program understanding and maintenance. The book includes a preface (header comments), chapters, a table of contents, and an index (cross-reference). It is generated semi-automatically by a program called *Bookmaker*. The authors mention two experiments that show that students using the book do maintenance tasks faster and with fewer errors.

This is a readable paper for students. A good student project might be organized around writing a version of the Bookmaker program for some specialized environment.

Oman90b

Oman, Paul. "Maintenance Tools." *IEEE Software 7,* 3 (May 1990), 59-65

This survey, part of the *IEEE Software* special "tools fair" issue, provides brief descriptions of several tools. The subsections are:

- Objective-C Browser details class structures
- Vifor transforms code skeletons to graphs
- Seels aids maintenance with code-block focus
- Battle Map, Act show code structure, complexity
- Grasp/Ada uses control structure
- Expert Dataflow and Static Analysis tool
- Surgeon's Assistant limits side effects
- Dependency Analysis Tool Set prototype

Ossher89

Ossher, Harold. "A Case Study in Structure Specification: A Grid Description of Scribe." *IEEE Transactions on Software Engineering 15,* 11 (Nov. 1989), 1397-1416.

Abstract: *The grid mechanism is a graphical notation for describing the structure of software systems and for specifying and enforcing structuring disciplines. It is intended to present complex structures in a clear and intuitive manner, yet it is formal: consistency between a system and a grid specification can be checked automatically. This enables one to examine a clear, graphical description of system structure with confidence that it accurately reflects the actual structure of the system.*

This paper describes a case study in which the grid mechanism was used to describe the structure of Scribe, a document processing system in widespread use. The structure description is presented and explained in some detail, and the effectiveness of the grid at specifying the important structural features of Scribe is discussed. Conclusions drawn from the case study include the following:

1) The grid succeeds in its objective of presenting complex structures clearly.
2) A grid specification forms a suitable basis for a narrative explanation of system structure.
3) Some detailed improvements would further enhance the expressiveness of the grid.
4) Environmental support is essential for serious use of the grid.

This is a good, but detailed, description of the grid mechanism and will probably be of most interest only to advanced students.

Rajlich88

Rajlich, Vaclav, Damaskinos, Nicholas, Linos, Panagiotis, Silva, Joao, and Khorshid, Wafa. “Visual Support for Programming-in-the-large.” *Proc. Conference on Software Maintenance - 1988*, Phoenix Arizona, IEEE Computer Society, October 1988, 92-99.

Abstract: *In this paper a brief description of the VIFOR (Visual Interactive FORtran) environment is given. VIFOR is based on a simple, but effective data model of Fortran programs. The model contains three entity classes and three relation classes only. Programs can be displayed and edited in two forms: the traditional one (i.e. code) and in the visual form. VIFOR contains transformation tools for both directions, i.e. from code to visual form and from visual form to skeletons of code. Hence it is suitable for reverse engineering and maintenance of existing code. Specially designed browsers implement the graphical interface.*

An example of the cross-reference database approach to tool construction, combined with a browsing interface.

Schwanke89

Schwanke, R. W., and Platoff, M. A. “Cross References are Features.” *Proc. ACM 2nd Intl. Workshop on Software Configuration Management,* Princeton, N. J., October 1989, 86-95.

This paper describes a clustering technique for software dependencies used in the ARCH “architect’s assistant” system being developed at Siemens. The method identifies components to be clustered based on their “shared neighbors.” For example, two subroutines will be clustered together if they are called by the same routines or if they call the same routines, instead of clustering together routines that call each other. Applications are shown for summarizing a call graph, splitting an include file, and improving modularity.

Taenzer89

Taenzer, David, Ganti, Murthy, Podar, Sunil. “Object-Oriented Software Reuse: The Yo-Yo Problem”. *Journal of Object-Oriented Programming 2,* 3 (Sept. 1989), 30-35.

This paper provides an interesting analysis of the complex dependencies created by polymorphism and method inheritance in object-oriented languages such as Objective-C. It may temper the reader’s enthusiasm for this particular software technology!

Tomayko89

Tomayko, J. E. “Teaching Maintenance Using Large Software Artifacts.” *Software Engineering Education. SEI Conference 1989*, Norman E. Gibbs, ed. *Lecture Notes in Computer Science 376*, Berlin: Springer-Verlag, 1989, 3-15.

Abstract: *A method for teaching software maintenance at the graduate level using software artifacts is described. Objectives, the syllabus, and assignments are included in annotated form. A discussion of the actual events and lessons learned in a prototype course is presented.*

Another paper useful for the instructor who is planning a maintenance-oriented student project.

Weiser81

Weiser, Mark. "Program Slicing." *Proc. 5th Intl. Conference on Software Engineering*, San Diego, California, IEEE Computer Society, March 1981, 439-449.

Abstract: *Program slicing is a method used by experienced computer programmers for abstracting from programs. Starting from a subset of a program's behavior, slicing reduces that program to a minimal form which still produces that behavior. The reduced program, called a "slice", is an independent program guaranteed to faithfully represent the original program within the domain of the specified subset of behavior.*

Finding a slice is generally unsolvable. A dataflow algorithm is presented for approximating slices when the behavior subset is specified as the values of a set of variables at a statement. Experimental evidence is presented that these slices are used by programmers during debugging. Experience with two automatic slicing tools is summarized. New measures of program complexity are suggested based on the organization of a program's slices.

This paper is the original exposition of program slicing and is readable both as a general introduction to slicing and as a detailed description of algorithms.

Weiser82

Weiser, Mark. "Programmers Use Slices When Debugging." *Comm. ACM 25*, 7 (July 1982), 446-452.

Abstract: *Computer programmers break apart large programs into smaller coherent pieces. Each of these pieces: functions, subroutines, modules, or abstract datatypes, is usually a contiguous piece of program text. The experiment reported here shows that programmers also routinely break programs into one kind of coherent piece which is not contiguous. When debugging unfamiliar programs programmers use program pieces called slices which are sets of statements related by their flow of data. The statements in a slice are not necessarily textually contiguous, but may be scattered through a program.*

Wilde89

Wilde, Norman, Huitt, Ross, and Huitt, Scott. "Dependency Analysis Tools: Reusable Components for Software Maintenance." *Proc. Conference on Software Maintenance - 1989,* Miami, Florida, IEEE Computer Society, October 1989, 126-131.

Abstract: *Software maintenance is costly because of the many complex interrelationships in a large software system; an understanding of these program dependencies is fundamental to efficient software change. This paper describes a general purpose toolset that is now being developed to capture and analyze software dependencies. The tools are designed to serve as reusable components. They may be used not only to aid programmers directly in understanding programs but also as a basis from which other specialized tools can be constructed.*

The tools use the concept of a dependency graph *as a basic abstraction to simplify the understanding of software relationships. Definitional, calling, functional and data-flow dependencies are analyzed. An* external dependency graph *for each function is developed to encapsulate the effects of function calls.*

Yau78

Yau, S. S., Collofello, J. S., and MacGregor, T. "Ripple Effect Analysis of Software Maintenance." *Proc. Compsac 78,* IEEE Computer Society, 1978, 60-65.

Abstract: *Maintenance of large-scale software systems is a complex and expensive process. Large-scale software systems often possess both a set of functional and performance requirements. Thus, it is important for maintenance personnel to consider the ramifications of a proposed program modification from both a functional and a performance perspective. In this paper the ripple effect which results as a consequence of program modification will be analyzed. A technique is developed to analyze this ripple effect from both functional and performance perspectives. A figure-of-merit is then proposed to estimate the complexity of program modification. This figure can be used as a basis upon which various modifications can be evaluated.*

This paper explains the basic concept of ripple analysis and shows how ripple effects can be calculated.

Yau84

Yau, Stephen S. *Methodology for Software Maintenance.* Report RADC-TR-83-262, Rome Air Development Center, Air Force Systems Command, Griffiss Air Force Base, N. Y., February 1984.

Abstract: *Improved techniques for specifying and implementing software modifications were developed including logical ripple effect analysis, logical and performance stability measures, and effective testing for software maintenance. An experiment was performed to analyze stability measurements.*

This is a quite extensive report on a project that involved constructing several kinds of maintenance tools.

Yau88

Yau, Stephen S., and Liu, Sying-Syang. *Some Approaches to Logical Ripple Effect Analysis.* Report SERC-TR-24-F, Software Engineering Research Center, Computer and Information Sciences Department, University of Florida, Gainesville, Fla., 1988.

This report presents improved algorithms for ripple effect analysis of a software change.

Zvegintzov89

Software Maintenance Tools - Release 2.0. Zvegintzov, Nicholas, editor. New York: Software Maintenance News, 1989.

This very good survey, which is updated from time to time, explains the use of maintenance tools and surveys many of the current commercial offerings.

The publication *Software Maintenance News* is also a good practitioner-oriented source on current developments in software maintenance.