Software Engineering
or
Methods for the Multi-Person Construction of Multi-Version Programs

Prof.Dr. D.L. Parnas
Technische Hochschule Darmstadt
Fachbereich Informatik, Betriebssysteme I
6100 Darmstadt, Steubenplatz 12

Abstract

This talk will describe some methods which have been used to produce a family of related software products using many relatively unskilled programmers. The primary topics of the talk will be:

1. An interpretation of the word "structure" with regard to software;
2. Criteria to be used in decomposing software into modules;
3. Problems and techniques with regard to software module specifications.

The talk will be introductory in nature emphasizing the desired properties of well engineered software systems and providing an overview of new methods which have proven useful in achieving those properties.

INTRODUCTION

The title of this paper is intended to suggest that software engineering is programming under at least one of the following two conditions:

(1) More than one person is involved in the construction and/or use of the program and
(2) more than one version of the program will be produced.

By the above I intend to emphasize the fact that software engineering is not confined to the production of certain classes of programs (e.g. compilers, operating systems, file systems), but is present whenever we are not in the situation of writing a program exclusively for our own use (solo programming).

In producing software we find three problems which are not significant in the solo programming situation:

(1) How to divide the job of producing the software among subgroups.
(2) How to specifiy to the user and to the various subgroups the exact behaviour demanded of each.
(3) How to communicate information about the occurrance of errors between the various program parts and to the user.

It should be clear that these problems are not present in the "solo-programming" situation and that classical programming textbooks do not treat them. The third problem is particularly intractable as even the latest in programming techniques proceed on the assumption that everything will go well.

In this paper we will briefly review some techniques which have been developed for software engineering situations, and the results of preliminary experiment to evaluate the value of those techniques. These results are not new, having been presented earlier in [1, 2, 3, 4, 5, 6]; they will be presented again as concisely as possible. The last section of this paper will present the result of more recent attempts to evaluate the techniques by applying them to the design of operating systems.

What is a Well Structured Program?

Before we can proceed we must explore the use of the word "structure" when discussing programs.

The word "structure" is used to refer to a partial description of a system. A structure description shows the system divided into a set of modules, and specifies some connections between the modules. Any given system admits many such descriptions. Since structure descriptions are not unique, our usage of "module" does not allow a precise definition (parallel to that of "subroutine" in software and of "card" in hardware). The definitions of the latter words delineate a restricted class of objects in a way that the definition of "module" does not. Nevertheless, "module" is useful in the same manner that "unit" is in military or economic discussions. We shall continue to use "module" without a precise definition. It refers to portions of a system indicated in a description of that system. A precise definition is not only system dependent, it is also dependent upon the particular description under discussion.

The term "connection" is usually accepted more readily. Many assume that the "connections" are control transfer points, passed parameters, and shared data for software, wires or other physical connections for hardware. Such a definition of "connection" is a highly dangerous oversimplification which results in misleading structure descriptions. The connections between modules are the assumptions which the modules make about each other. In most systems we find that these connections are much more extensive than the calling sequences and control block formats usually shown in system structure descriptions.

The meaning of the above remark can be exhibited by considering two situations in which the structure of a system is terribly important: (1) making of changes in a system, and (2) proving system correctness. (I feel no need to argue the necessity of proving programs correct, or to support the necessity of making changes. I wish to use those hypothetical situations to exhibit the meaning of "connection").

Correctness proofs for programs can become so complex that their own correctness is in question (e.g. [7], [8]). For large systems we must make use of the structure of the programs in producing the proofs. We must examine the programs comprising each

module separately. For each module we will identify (1) the system properties that it is expected to guarantee, and (2) the properties which it expects of other modules. The correctness proof for each module will take (1) as the set of theorems to be proven and (2) as a set of axioms which may be used in proving that the programs do indeed guarantee the truths of the theorems. Eventually the theorems proven about each module will be used in proving the correctness of the whole system. The task of proving system correctness will be facilitated by this process if and only if the amount of information in the statement sets (1) and (2) is significantly less than the information in the complete description of the programs which implement the modules.

We now consider making a change in the completed system. We ask, "What changes can be made to one module without involving change to other modules?" We may make only those changes which do not violate the assumptions which other modules make about the module being changed. In other words, a single module may be changed only while the "connections" still "fit".

In both cases we have a strong argument for making the connections contain as little information as possible. Systems in which the connections between modules contain little information are labeled well structured.

Two Techniques for Controlling the Structure of Systems Programs

In studying the problem of producing well structured systems programs we have discovered that there are two basic functions which the designer must perform carefully in order to control the structure of his programs. The first of these is the division of the project into modules or work assignments (decomposition); the second function is precise specification of those modules.

Some informal experiments have revealed that there is a remarkable consistency in the way that programmers will divide systems into modules. For example, I have repeatedly asked programmers how they would go about dividing the project of pro-

ducing a KWIC index program into work assignments. With very rare exceptions the programmers suggested a decomposition based on a flowchart description of the whole system. They are following the lessons that they received in their early training in programming. Programmers are almost invariably taught that the first step in producing a program is to write a "rough" flowchart and then proceed to detail each of the boxes in it. This is often an excellent strategy for a "solo" programming project, but, as demonstrated in [2] it is seldom a good procedure for dividing a project into work assignments. The conlusion of the previous section was that a well structured program was one with minimal interconnections between its structure components or modules. Because information must be passed in large chunks (and using fixed conventions) between the various phases (boxes in the flowchart) the conventional approach results in modules which have quite strong interconnections. As was indicated in [2], systems which result from such a design are quite difficult to change in any significant way. We can, however, forget our flowcharts and attempt instead to define our modules "around" assumptions which are likely to change. One then designs a module which "hides" or contains each one. Such modules have rather abstract interfaces which are relatively unlikely to change. If we succeed in defining interfaces which are sufficiently "solid" (i.e., they will not change), changes in the system can be confined to one module. The reader is referred to [2] for a deeper discussion of this point including a detailed example. Teaching experience described in [9] has shown that, at this point in time, such skills can only be taught by example providing simulated experience.

The second function of the designer is that of specification. There is good reason to believe [6] that the designer can obtain a system with the structure he suggested only if he has a way of precisely defining which assumptions the designers of one module are permitted to make about other modules with which they interface. In describing those assumptions (writing specifications for each module), the designer is walking a tightrope. If the programmer has too little information about the other modules, he will be unable to produce an efficiently working

system. The need for sufficient information is obvious to almost everyone. The surprising side of the tightrope is that it is also wrong to provide too much information. If the excess information is used (i.e., if additional assumptions are made), the structure of the program will not be that intended by the designer. The most common approach to software module specification is to reveal a rough description of the internal structure of the module. The next most common approach is to reveal a description of a "hypothetical" implementation of the module. Almost invariably we are told the structure of some real or imagined table (or procedure) which the user of the module does not have access to. Both of these approaches are fraught with danger. In the first, the system may become hard to change if the correctness of the interfacing programs depends upon the internal information which was revealed. In the second case the program may never work correctly, because the correctness depended on some of the "hypothetical" implementation which was never true. It is oberservable that it is extremely difficult to reveal some part of an implementation in order to be precise, and then instruct the reader precisely which parts of the reavealed information he must not use.

We are now gaining some experience with a specification technique which takes as its goal the precise specification of externally visible aspects without suggesting internal constructions. The approach is to specify identities or relations between the externally visible aspects of the module rather than reveal the internal construction. Thus the specifications relate the externally visible functions to each other rather than to some real or imagined lower level machine. In syntax these specifications resemble programs but the refusal to mention lower level or internal mechanisms distinguishes them from other forms of specifications or programs. The reader is referred to [3] for more discussion and detailed examples.

Results

On the basis of the above considerations we have obtained quite satisfactory results in small scale experiments with undergraduate classes. For example, in the fall of 1971 we produced 192 quite distinct versions of a KWIC index program using 15 modules

produced by 15 students. (We set out to produce 45 versions using 2o students, but five of the students produced modules which failed to meet specifications.) Of the 15 which passed the preliminary testing we could make 192 distinct combinations. We selected only 25 of these for testing (for economy reasons), but all of the 25 ran successfully. All students worked independently and had no advance knowledge of the combinations which would be tested. The actual testing was carried out by graduate students with no knowledge of the internal behaviour of any program module and who did not alter the internals (except to reduce excessive space requests). While we would prefer to have tested the techniques on a larger and more interesting project, we feel that the limited use suggests that it is both feasible and valuable to produce systems using the principles outlined above.

Error handling

The treatment of run time errors is made more difficult by the information hiding approach to structuring programs. When an error is detected, the information about what has gone wrong is expressed in terms of data structures and programs which are not known to the majority of the system. The information needed to understand the cause of the error and the procedure for corrective action is likely to be in other modules. If the information about the error is communicated in terms of the hidden program structures, the structure of the resulting system will be destroyed by the distribution of the additional assumptions.

In [1o] an approach to solving this problem has been outlined. This approach was used, in a primitive form, in the experiment described above. Even in this form it had the advantage that, when an error was discovered, it took no detailed knowledge of any module to identify which module had caused the error. This was a great advantage in managing the project and a complete change from the author's experience in other multi-person projects. Identifying the module at fault is often the major problem in correcting an error.

Error recovery as discussed in [1o] has not yet been attempted so we can report no experimental results. Study of the scheme has shown that it leads to difficulties in maintenance of the call stacks and an improved method is now under study.

More Recent Work

Since it is well known that any new method works well when applied by enthusiastic researchers to a small problem, we have decided to continue our study by applying the techniques discussed above to the production of a more realistic family of programs. In particular, we are now applying the technique to the specification of the modules which make up a family of operating systems, which we hope will be suitable for a broad set of machines in a broad range of applications. In principle, one can consider the machine dependent aspects of any operating system to be the implementation details to be hidden within the various modules of the system.

The first step was specification of a mechanism which allows all programs to run without using physical addresses. This work was reported by Price [11] and Price and Parnas [12].

As a result we have discovered some unexpected difficulties:
(1) The form of specification used on the small examples requires considerable change before it is practical for more realistic problems. The specification makes Price's module appear far more complex than it is. The information density is much too low. We are now studying new methods of specification so that the specifications for realistic modules will be of realistic size.

(2) In producing the specification of the Price module, we found it necessary to violate one of our cardinal rules about specification. The specification is produced in terms of some "hidden" functions, functions which are not accessable to the user. We are unhappy about this procedure, the specification should not refer to facts the user cannot detect. Price had to introduce the hidden functions, because each attempt to work without them made the specification longer. In some current work we are now developing extentions to our specification techniques, which

make it possible to remove the hidden functions without increasing the length of the specification. The hidden functions can be replaced by sets which characterize the history of the module from an external view.

(3) Probably the most important insight to arise out of the Price work is that we have found fundamental limitations to the ideas mentioned above. In plain English, not everything can be hidden! An example centers about the treatment of I/O in our operating system family. Because our initial implementation was on the PDP/11, a machine without explicit I/O instructions, we found that no special consideration of I/O was needed in the design of the lower levels. Now that we are studying the implementation of these concepts on a /360-like machine, we find that "hiding" the existence of the I/O instructions will introduce inefficiency. While we see the existence of I/O instructions on our machine as a disadvantage, POPEK [13] attempting to transfer ideas developed on a /360 to a PDP/11 sees the lack of I/O as a disadvantage on the PDP/11. At the moment we have not found a system structure which is really suitable for both types of machines.

A similar situation arose because of the PDP/11/45's ability to use seperate address maps for data and instructions. Had we taken advantage of this "feature" in our design, the resulting design would be impractical for other machines.

Conclusion

It seems clear that the information hiding principle espoused earlier in this paper is a valuable technique for software engineers, but we have definitely found limitations. The purpose of our further research is then

(1) to apply those techniques to produce information about good structures for frequently built items such as operating systems,

(2) to develop techniques for extending the usefulness of the concepts and understanding the real limits in practical use.

References

[1] Parnas, D.L., "Some Conclusions from an Experiment in Software Engineering", Proceedings of the 1972 FJCC.

[2] Parnas, D.L., "On the Criteria to be Used in Decomposing Systems into Modules", Communications of the ACM (Programming Techniques Department), Dec. 1972.

[3] Parnas, D.L., "A Technique for Software Module Specification with Examples", Communications of the ACM (Programming Techniques Department), May 1972.

[4] L. Robinson and D.L. Parnas, "A Program Holder Module", Technical Report, Carnegie-Mellon University, June 1973.

[5] Robinson, L., "Design and Implementation of A Multi-Level System Using Software Modules", Technical Report, Carnegie-Mellon University, June 1973.

[6] Parnas, D.L., "Information Distribution Aspects of Design Methodology", Proceedings of IFIP Congress 1971.

[7] Balzer, Robert M., "Studies Concerning Minimal Time Solutions to the Firing Squad Synchronization Problem", Ph.D. Thesis, Carnegie Institute of Technology, 1966.

[8] London, R., "Certification of Treesort 3", CACM, June 197o.

[9] Parnas, D.L., "A Course on Software Engineering Techniques", included in the Proceedings of the ACM SIGCSE, Second Technical Symposium, March 24-25, 1972.

[1o] Parnas, D.L., "On the Response to Detected Errors in Hierarchically Structured Systems", Technical Report, Carnegie-Mellon University, 1972.

[11] Price, W.R., "Implications of a Virtual Memory Mechanism for Implementing Protection in a Family of Operating Systems", Technical Report (Ph.D. Thesis), Carnegie-Mellon University, June 1973.

[12] Parnas, D.L., Price, W.R., "The Design of the Virtual Memory Aspects of a Virtual Machine", Proceedings of the ACM SIGARCH-SIGOPS Workshop on Virtual Computer Systems, March 1973.

[13] Popek, G.J. and Kline, C., "Verifiable Secure Operating Systems Software", AFIPS Conference Proceedings, 1974, NCC AFIPS Press, Montvale, N.J. U.S.A.