THE INFLUENCE OF SOFTWARE STRUCTURE ON RELIABILITY

D.L. Parnas
Research Group on
Operating System Structure
Technical University
Darmstadt
Darmstadt, West Germany

Keywords: Reliability, Software Structure, Modules, Interfaces, Error Handling, Error Correction

Abstract

This paper assumes software structure to be characterized by the interfaces between subsystems or modules. Reliability is considered to be a measure of the extent to which the system can be expected to deliver usable services when those services are demanded. It is argued that reliability and correctness (in the sense used in current computer literature) are not synonyms. The differences suggest techniques by which the reliability of software can be improved even while the production of correct software remains beyond our reach. In general, the techniques involve considering certain unpleasant facts of life at an early stage in the design process, the stage where the structure is determined, rather than later. An appendix gives some specific examples of questions which, if they are thoughtfully considered early in the design, can lead to more reliable systems.

Introduction

The word structure is used here to indicate the way that a software system is divided into modules and the assumptions that the various modules make about each other. A module here is intended to be a unit of work assignment to groups of programmers or individual programmers. It is intended to be a unit which can be constructed with no knowledge of the internal structure of other modules. It is also intended that most design decisions in a system can be changed by altering a single module. More detailed descriptions of this interpretation of "structure" can be found in [1,2]. The concrete representations of the structure of a system should be the set of module specifications.

In this paper we intend to make a strong distinction between "reliability" and "correctness" as a static property of a piece of software and its specification. If a piece of software meets its specification, it is correct - if it does not, it is incorrect. Reliability, in contrast, is a statistical measure relating a system to the pattern of demands we make upon it. We consider a system to be highly reliable, if it is highly probable that, when we demand a service from the system, it will perform to our satisfaction. Often, we weigh situations in which we urgently need the services of the system more highly than those situations in which our demands are routine. Thus a system may be considered unreliable if it usually fails in emergency situations even though these situations amount to a small percentage of the total usage of the system.

Much of the software that we now consider reliable is actually not correct. We may consider the system reliable in spite of the errors if either:

(1) The programming errors do not make the system unusable (e.g. format of output, erroneous output which is easily detected and corrected by the user) or

(2) The situations in which the errors have an effect do not occur very often and the situations are not especially likely to occur, at moments when the need for the system is very great.

On the other hand, "correct" (in the sense of mathematically verified)software can be quite unreliable. This happens, when the specification for the software is not a complete description of what we expect from the software specifications. Correctness proofs are generally based on the (often unstated) assumptions that (a) the machine which interprets the software will function perfectly, and that (b) all data inputs to the system will be correct. If these assumptions are false a "correct" system can fail when we make a demand of it. The system could be proven "correct" only because the specification (which we used in proving correctness) did not state any requirements about the behavior of the system in circumstances when either of the abovementioned assumptions was not valid. Such failures have an unusually great effect on our estimation of the reliability of the system, because errors in input and errors by the supporting machine are especially likely in moments of stress.

The obvious ways that the structure of a system adversely affects its reliability are those ways that the structure affects the correctness of the system too. It is widely recognized that badly structured systems are more likely to be incorrect and that, when they must be changed, the changes are more likely to introduce errors. In this paper I wish to discuss some of the less obvious influences of software structure on reliability. These are situations in which we see differences between "reliability" and "correctness". We consider a structure to be "bad" if many modules are based on assumptions which are likely to become false. Particularly relevant to reliability are structures where many modules are dessigned on the assumption that "nothing will be wrong". Specifcally:

(1) The software structure is based upon the assumption that everything outside the software will behave correctly;

(2) The structure is based on the assumption that there are no software errors;

(3) The structure is based upon an "all or nothing" approach. There is no definition of degrees of imperfect behavior.

The tangible description of the structure of a system is the specification of the intermodule interfaces. Below we discuss how these interfaces must appear if the structure is not to have the problems listed above.

On incorrect external influences

The interfaces between the modules must enable the communication of information about external errors. For example, it should be possible for a module to be informed that information, given to it earlier, was incorrect, or that a request, which it had issued some time ago, was executed incorrectly. It should be possible for a module, which detects inconsistancies in incoming information, to inform the supplying module about those inconsistancies. The module supplying those data should be designed to respond meaningfully to such a notification.

In most "real" systems the intermodule interfaces are defined in such a way that the responsibility for detecting and responding to external errors is "left" to those modules that deal directly with the external interfaces. For example responsibility for noticing inconsistancies in operator input (e.g. an alphabetic character in a numeric field) is given to the program with responsibility for handling the operator's console. Another example, responsibility for recovering from an error in memory hardware is usually given to "low level" routines. This appears to conform to our intuitive notions of modularity, but in fact it is a direct violation of the "information hiding principle" used in dividing programs into modules [1,2].

An attempt to confine error handling to a single module or level in a system leads to inflexible systems and systems which do not do as much error detection and correction as they could. The reasons:

(1) The information necessary for good recovery from most hardware errors is not usually present in the "hardware near" programs but in programs responsible for the more global strategies and techniques used in the system.

(2) The information necessary to detect an input error and to inform the operator of the difficulty is usually contained in modules responsible for storage of previously processed inputs - not in the modules responsible for direct handling the console interface.

The solution to these problems lies in specifying module interfaces which allow communication about errors between the modules. In most cases the module detecting an error should not be the module which corrects it. The module which determines the correction strategy will often require many other submodules to take corrective action as well. Unfortunately, in most of the real systems that I have looked at, consideration of these problems comes late in the design process (usually during use) and the module interfaces must be violated in order to obtain the desired behavior.

Some aspects of module design related to inter-module error communication have been discussed and illustrated in [5] and further examples are given in an appendix to this paper.

On Incorrect Software

The wave of interest in structured programming has brought with it a new wave of optimism. Some people apparently believe that good intentions and mathematical rigor will lead to the production of correct programs. Such an approach seems to ignore all the errors that good-willed, rigorous mathematicians have made in the past. Even if we will eventually have error-free programs, we must consider the transition period. During that time the structure of the system should not be based upon the assumption that there are no errors in the individual modules, but allow for the possible malfunctioning of software components due to internal error. The module specifications should give each of the modules the responsibility of making certain basic checks on the behavior of those modules with which it interacts. Even if the software were written correctly, earlier hardware errors might have made it incorrect.

The literature now contains a number of examples of "well-structured" systems which have the property that all the components are written with no provision for the malfunctioning of other components. A design with such a structure will be less reliable than one which requires each of the components to include provisions for the most likely malfunctioning of the others.

The "all or nothing" Assumption

The predicates usually used in proving the correctness of programs define the behavior, which we consider the desired behavior of the system under ideal circumstances. Absolutely nothing is said about what should be done if that behavior cannot be obtained. Using such a set of predicates as a specification can result in a system which behaves perfectly when everything goes well, but behaves randomly in other situations. An alternative would be to include several sets of predicates, each set defining a "degree" of desired behavior. Degree 0 would define the "ideal" behavior and correspond to the predicates currently used to prove correctness. Degree i+1 would be a set of requirements which we would like to be satisfied in situations where the requirements for degree i cannot be met. Such a system gives some guidance to the programmers about what the program should do when something goes wrong. The information is particularly necessary

for writing the error handling routines. The inclusion of such predicates does not simplify the correctness proof - but it can lead to more reliable systems.
In many applications users would consider a system which behaved perfectly under ideal circumstances but randomly at all other times to be less reliable than a system which was not loo% "correct", but provided some services even when it could not function perfectly.

The effect of unrealistic software structures on reliability

"Structures" must often be understood as a euphemism for "restricted". When we define a system structure, we restrict the ways in which the components may interact. We consider a structure to be unrealistic when it does not permit the inter-module interactions which are necessary to meet realistic goals (e.g. as meaningful response to meaningless inputs and efficiency).

When a structure is unrealistic it contributes to unreliability in two ways:

(1) It does not allow certain error recovery procedures which could lead to increased reliability;
(2) It encourages practical programmers to violate the restrictions set down by the structure in order to meet realistic goals. This results in so called "unstructured systems" which are more accurately described simply as "poorly structured systems" or systems in which many assumptions about details, which are likely to change, are widely distributed.

From these considerations I feel forced to state my opinion that some of the suggestions for system structuring which have appeared in the literature could actually have a negative effect on reliability if applied. For example, many of the suggestions for operating system structuring achieve an appealing simplicity by assuming by assuming either that (1) relatively long programs can be carried out in a machine state with interrupts disabled, or (2) the number of processes in a system or and their speeds are such that there will rarely be more than one process waiting for entrance to a critical section. If these assumptions are realistic, the simplifications that result from them are clearly beneficial. Unfortunately, for many of the systems which the practitioners in this audience deal with, the assumptions are unrealistic. Even if the assumptions are true when the system structure is conceived, by the time that the system actually sees use, the application requirements may have been changed so that the assumptions are no longer valid. Note that the assumptions, that one is in a critical section for only a short time is very questionable when one considers the possibility of an error occurring during execution of the critical sections. If only one critical section is used and error handling operates in the same section, a deadlock may result. The designer who chooses to base a system on such simplifying assumptions must weigh the benefit that comes from the simplicity against the estimated cost that would be incurred if the assumptions proved invalid.
These considerations point out the danger in the wide spread assumption that simpler programs are "better structured" than complex ones. Careful attention to program structure usually brings simplicity and clarity as a result, but simplicity can also be obtained by neglecting essential problems.

Summary and Conclusions

The above considerations have led me to the conclusion that there are two quite distinct and complementary approaches that one can take to the problem of obtaining more reliable software. In the first approach one regards "unreliable" as a euphemism for "error ridden" and we develop techniques for (1) writing correct programs in the first place and (2) verifying that programs are correct or finding their errors. The second approach is based on the observation that with computer systems (just as with people) "reliable" and "correct" are not synonyms. This leads us to suggest steps in the design process which can lead to an improvement in reliability without necessarily affecting "correctness". These are:

(1) When writing specifications for the system and each of the modules, be realistic enough to specify the behavior which is desired when perfect behavior is not obtainable.
(2) In specifying the interface between the system and its environment and the interfaces between the various sub-systems modules, require the programs to be suspicious. Specify not only what the interfacing elements should do in the normal case, but also which assumptions should be verified by run-time checks and which actions are required when an error is detected.
(3) Include in the interfaces conventions for informing affected modules about things that have gone wrong elsewhere in the system.

It is important that these actions be taken at early stages in the design of a system. Usually those assumptions made at the early stages of the system are most costly to change at later stages. Every assumption which is made in the design of an interface affects more than one module and generally interface decisions which are made early in the system design. Realism from the start is to be preferred over expensive changes when the system reaches "puberty" (the stage before maturity when the facts of life become apparent).

I hope however that the reader will not reach the conclusion that one should abandon or ignore the attempts to write correct programs and automate the verification process. As I stated at the start of this section the two approaches are complementary. Nobody favors incorrect software.

I have chosen to emphasize the second approach for two reasons:

(1) I believe enough has been said and written about the first and that I could not add much to it.
(2) I personally believe that for payoff in the next 5-1o years the second approach (a) is more promising and (b) is not receiving enough attention. Although consideration of the points mentioned above, is clearly within the intellectual reach of those responsible for specifying and designing systems today, and I have looked at many specifications for real and academic systems as well, I have yet to find one which makes significant efforts in the directions I have outlined.

In the appendix of this paper I have included a few illustrative examples in the hope of making the rather general principles outlined above more concrete for those who would like to apply them in practice.

Appendix

Examples of System and Module Requirements intended to make Real Software more reliable.

In this example we consider the early stages of the design of a communications support system. The purpose of this system is to (1) provide for the effective use of the communications equipment according to established priorities, (2) provide a convenient interface for the communications operators (those who type the messages in), and (3) provide a data bank from which information about messages sent and received can be retrieved.

Below we give a list of questions which we believe should be considered during the early stages of the system design. Note that we have said "considered" and not " answered". Many of the questions may not be best answered at an early stage because the correct answer may differ from installation to installation or even change during the life cycle of the system. By considering such questions however one can so structure the system that it is possible to "answer" the question at a later stage by writing or changing a single program rather than by redesigning vast portions of the system.

I. Questions which should be considered when designing the interface to the system

(1) If a piece of communications equipment fails during a transmission - how should the system be informed of that failure? How much information about the failure will be supplied? How should the system respond?
(2) What action should the system take if an operator inputs a message with a priority beyond his privileges? How can an operator indicate that he made an error in priority?
(3) Assume that the system keeps message logs including a separate log of absolute top priority messages. What corrective action should the system take if it is determined that the mass storage device used for keeping the log was defective and the information stored on it has been lost? +
(4) How can one inform the system (or how can the system detect) the failure or misfunctioning of a part of the main memory?

II. Questions which should be considered when defining the intermodule interfaces.

Here we assume that the system uses a memory allocation module which allocates blocks of 512 words whenever requested to supply such by other modules who call GETSPACE. We also assume that there is a deadlock prevention module (the banker) which approves or disapproves requests for all resource allocation modules without knowing any details about the individual resources.

(1) Does the interface to the memory allocation module allow it to be informed that a part of the memory is malfunctioning and should not be allocated?
(2) What should the memory allocation module do if it is informed that a memory area which is already allocated to a program is not functioning properly.
(3) Does the Banker module contain programs which allow it to be informed of a sudden failure of a resource such as a memory bank? What action should it take if it discovers that a deadlock is the result of this failure?
(4) How can a program which is using a piece of defective memory be informed of this problem? How should that program react?
(5) If the result of being allocated non-functioning memory is that the program process is waiting and it will not continue until the memory is repaired, how should one inform the memory allocation module and the banker of this problem? (Both of those modules usually proceed on the assumption that the program will eventually return its resources - that assumption is now false.
(6) If it is discovered that one program is misfunctioning and continues to ask for resources without returning them, how should the above named modules be informed of this problem? How should they react?

+ Possible Answer:
(1): try to reconstruct the data from other logs, if that's impossible, then
(2): if part of the data has been lost, ask the operator to resupply. If (1) + (2) not possible, then...
(N): if it is not possible to maintain a log with the storage available, produce a punched card version or paper tape version.

Comments on the above questions:

It is possible to generate an arbitrarily long list of such questions, starting with highly likely situations such as those mentioned above and ending at some point where the problems mentioned are deemed so unlikely that it is not worthwhile preparing for them.

I have examined descriptions of a wide variety of systems ranging from "model systems" produced by small groups of highly skilled people in Universities to "real" systems which appeared to have been produced by a stream of visitors who never met each other, but managed somehow to glue something together which functions. As far as I, (an outsider,) could tell questions such as these are never considered at the design stage in which the external and intermodule specifications are determined. In other words, the structure is based upon the assumption that such situations will never occurr.

When reality forces the developers to consider such situations later in the design, the necessary changes violate the early assumptions and hence are expensive and time consuming to introduce and often lead to many difficult to find bugs.

My thesis is that consideration of such questions when designing the structure of a system is within the state of the art and carries a high short range payoff which can be measured in terms of increased system reliability.

Acknowledgments

Conversations with A. Endres of IBM Germany and H. Würges of the Technical University, Darmstadt, have contributed greatly to my understanding of the problems discussed in this paper. Remarks by W. Bartussek on an early draft of this paper have been very helpful.

References

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

[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., "Some Conclusions from an Experiment in Software Engineering", Proceedings of the 1972 FJCC.

[4] Kaiser, C., Krakowiak, S., "An Analysis of Some Run-Time Errors in an Operating System", Aspects Théoriques et Pratiques des Systèmes d'Exploitation, IRIA, Rocquencourt, April 1974.

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