MASSACHUSETTS INSTITUTE OF TECHNOLOGY

PROJECT MAC

Computation Structures Group Memo 69

The Design and Construction of Software Systems

by

Jack B. Dennis

Notes Prepared for an Advanced Course on Software Engineering,
Technical University of Munich, February 1972

June 1972

This research was done at Project MAC, MIT, and was supported in part by the Advanced Research Projects Agency, Department of Defense, under Office of Naval Research Contract Nonr N00014-70-A-0362, and in part by the National Science Foundation under grant GJ-432.

The Design and Construction of Software Systems

1. INTRODUCTION

Software Engineering is the application of principles, skills and art to the design and construction of programs and systems of programs. It is often asserted that software engineering is largely art and based very little on sound principle. Yet trends are visible and new ideas are developing that promise to substantially increase the role of theory and principle in the design and construction of software systems. In this lecture, I wish to present a frame of reference for relating the material to be presented in this course. In addition, I shall try to assess the limitations of known principles for the practical needs of software engineering, and the prospects for broad future application of principle to the design and construction of software. This sketch will certainly be a very personal view of the field, for there is little published material that attempts to characterize software engineering.

The theme of this talk is that behind the absence of a satisfactory set of principles for the practice of software engineering lies the lack of adequate means for representing software and hardware system designs. Further development of the theoretical foundation for programming language semantics and system representation is required to overcome the limitations of contemporary software engineering.

2. TERMINOLOGY

In presenting a framework for discussing principles of software engineering we immediately encounter problems of terminology: What is "software"? What do we mean by "computer system"?

2.1 COMPUTER SYSTEMS

We shall use the term computer system to mean a combination of hardware and software components that provides a definite form of service to a group of "users". A particular computer installation appears as many different computer systems depending on the group of users considered. For example, in a general purpose computer installation that offers the ability to edit and interpret programs expressed in the language Basic [1], we can identify at least three distinct computer systems and corresponding user groups:

system	user group
1. the computer hardware	operating system implementers
2. hardware plus operating system	subsystem implementers
3. hardware, operating system and Basic language subsystem	users of Basic

Any computer system defines a language in terms of which all software run on the computer system is expressed. I mean this in a very exact sense: A computer system provides representations for certain data types and information structures, and implements a set of primitive operations on these data types and structures. Let us consider the three cases mentioned above.

Suppose the computer system consists only of hardware (a processing unit and main memory, say). Then the data types correspond to the interpretations of memory words that are implicit in the built-in operations of the processor -- usually fixed and floating point representations for numerical quantities. In the absence of base registers in the processor, the information structures of this computer system are simply all possible contents of the main memory, selection of a desired component of a structure being accomplished through indexing or address computation. The effect of the interrupt feature of the computer must also be modelled in the language. The possibility of asynchronous interrupts makes the language defined by a hardware computer system nondeterministic; that is, there may be many successor states possible for a given state of the system.

When the central hardware is augmented by peripheral devices and an operating system, additional data types and classes of information structures are represented, new primitive operations are defined, and some features of the hardware are made inaccessible. One important addition is the availability of files as a representation for information structures -- data and programs. Separate address spaces are provided for each concurrent computation and a generalized means of referencing data items and programs is implemented. The absolute addressing mechanism of the hardware is often not available to the user. Similarly, the hardware facilities for process switching and interrupt processing are replaced by software primitives for interprocess communication, which are implemented by the scheduling modules of the operating system.

The operations and data structures of the language defined by hardware and operating system may be complex. For example, in this view, the action of a program linking loader must be considered as a primitive operation that transforms one information structure (representing a set of program modules generated by compilers) into a new information structure (a set of procedures linked together and assigned to the address space of a computation).

The inclusion of peripheral devices may alter the view the user has of the language of the computer system. In the absence of peripherals, the machine appears as a device into which one puts programs for execution. The language of the computer system is then the set of programs that can be represented in memory according to the computer system's instruction code. If users interact with a computer system from peripheral terminals, the system behaves as a device having a set of internal configurations and which responds to messages with answers depending on its extant configuration. The language of the system now appears to the user as a set of meaningful messages together with corresponding state transitions and conditioned responses.

Adding a software subsystem for the Basic programming language yields a third computer system. The language defined by it is a model for the commands and responses by which one interacts with the Basic subsystem from a user's terminal. In this language, the primitive data types and operations are those of Basic. Users have access to the language of the operating system only through use of the subsystem.

2.2 SOFTWARE SYSTEMS

The environment for a program consists of the computer system on which the program runs, together with any hardware components, other than those of the computer system, required by the program. By the term software system we mean the software and hardware components that must be added to a specific computer system, called the host system, in order to realize some desired function.

We may illustrate in terms of the examples cited above: For an operating system the host system may be the processing units and main memory hardware. The operating system is then a software system having many software modules and appropriate mass storage devices to hold files. This computer system may then serve as the host system for a software system that implements the language Basic. This software system consists of an editor, an interpreter, and a command processor. If the host system does not include a communications line controller, the implementer of Basic would find it necessary to add one to the host as part of the new computer system.

2.3 HIERARCHY

Hierarchical relationships occur in many forms in computer systems. Here, we will discuss just one form of hierarchy: the hierarchy of linguistic levels defined by successive layers of software. Each level of this hierarchy is a computer system characterized by the data types and primitive operations of its language. Each level is (or, is potentially) the host system for the definition of new linguistic levels through the addition of further software systems.

Hierarchy is a tool of software engineering which, if properly used, permits the components of several levels to be designed and developed separately. Of course, separate development of system levels is only possible if the languages corresponding to the boundaries between layers of software have been precisely specified and agreed to. For success, the implementers of a software system should not find it necessary to alter any component of the host system.

Such need would expose incompleteness or inefficiency of the host language for the objectives of the software system. This principle is often violated in practice, for example, when an inner layer of an operating system is modified so an accounting procedure may be implemented within an outer software layer.

We distinguish three techniques used to define a new linguistic level by a software system: _extension_, _translation_ and _interpretation_. Often combinations of the three techniques are used.

1. _Extension_: In defining a new linguistic level by procedural extension, the software system added to the host system is simply a collection of procedures that express the primitive operations of the new level in terms of the primitive operations of the host system. New data types or structure classes are implemented in this way and made available to users of the extended system in addition to the primitives and data types of the host system. In using extension the internal representations for procedures at both levels, host and new, are identical, syntactically and semantically.

2. _Translation_: Defining a new linguistic level by translation consists of writing a compiler to run on the host system that translates programs at the new linguistic level into programs in the language of the host system. The necessity of compilation as a step in running a program is characteristic of this technique. Representations of programs expressed in the language of the new level are not directly executed.

3. _Interpretation_: Defining a new linguistic level by interpretation consists of writing an interpreter for the language of the new level in terms of the data types and primitive operations of the host system. Programs at the new linguistic level are represented in directly executable form.

A software system may be designed so that all users of the host system are required to do so at the linguistic level of the software system. An example is a computer run under a specific operating system which all users of the computer must use. Alternatively, several software systems may share the same host, as in the case that several programming language systems operate under the same executive control program. Further, the definition of a new level may or may not deny the user access to part or all of the linguistic features of the host. The difference between use of extension and interpretation is that in extension the primitives of the host system remain available at the new level whereas this is usually not the case when interpretation is used.

It would seem that procedural extension ought not be considered as defining a new linguistic level unless the added procedures are grouped in a way that hierarchical relations are defined. Examples are the use of the technique for application packages and in the implementation of command languages of operating systems. In these cases, a collection of procedures defines the new linguistic level. Users of the new level are often prevented from using procedures outside the collection, and then the collection of procedures is essentially an interpreter for the new linguistic level.

Translation and interpretation are fundamentally different in the following respect: Two compilers for different source languages, if implemented for the same host system, produce compiled procedures in the language of the host. If standard procedure interfacing conventions of the host are honored by both compilers, then programs expressed in the two source languages may be operated together successfully. In contrast, interpretation is usually done because of a need to utilize a fundamentally different form of data organization from the host, or to obtain program monitoring and control features not possible at the host level. That is, the host level is incomplete for the objectives of the software system. If interpreters for two source languages are written in the language of the host, then communication between procedures expressed in the two languages will be difficult if not impossible, unless carefully coordinated planning is done by the implementers. Each interpreter will likely use entirely different representations for equivalent data types, hence each call on a procedure expressed in the other language would have to cause switching of interpreters and translation of all data to be communicated.

2.4 SYSTEM AND APPLICATION SOFTWARE

Traditionally "system program" refers to the layers of software that "belong" to a computer installation and are available to all clients of the installation; "application software" refers to the software brought to an installation by a client for performing his desired computation. This distinction between system and application software has lost meaning with the evolution of more sophisticated uses of computer systems. For example, one client of an installation may implement a new programming language and make it available to other clients of the installation. Or an installation may be devoted entirely to a particular application as in the case of real-time systems such as reservation and inventory systems.

Nevertheless, by using the concepts and terminology discussed above, we may list certain distinguishing characteristics that will serve to crudely classify software as system software or application software for the purposes of subsequent discussion.

system software: A collection of system programs usually forms a hierarchy of software systems having these properties:

1. The collection of programs are implemented under one authority.
2. The hierarchy of software systems defines a single linguistic level which applies to all users of the collection of programs.
3. Inner linguistic levels of the hierarchy are hidden from the user.
4. The outer linguistic level of the hierarchy is "complete" for the goals of the implementing authority.
5. The primary means of defining new linguistic levels is partial interpretation.

application software: An application program or software system usually has these properties:

1. The programs are expressed in terms of a "complete" linguistic level.
2. The programs define a new linguistic level by extension, translation, interpretation, or by some combination of these techniques.
3. The linguistic level defined by the program or software system is inadequate for defining further linguistic levels.
4. A variety of such programs or software systems are available to clients of an installation, and are often implemented under different authorities.

3. DESCRIPTION OF SOFTWARE SYSTEMS

The design and construction of a software system is, fundamentally, the creation of a complete and precise description of the system. The description of a software system is a collection of descriptions of its software and hardware components.

The complete and precise description of a software component is in reality a program expressed in a well-defined programming language. If this language is the language of the host system, or the translation of the program to the linguistic level defined by the host is strictly a clerical operation, then preparing the program completes the process of constructing the system component. Otherwise implementation of the component is incomplete until a correct representation of the component is prepared at the linguistic level of the host system.

In the case of a hardware component, a description is adequate only if it permits the designer of the software system to determine exactly the relevant behavior of the component for all situations that may occur during operation of the software system. Statements of interfacing conventions are insufficient, for these do not describe the function performed by the hardware component. Usually, an adequate description must take the form of a model of the internal operation of the component.

Besides descriptions of its hardware and software components, two further descriptions are required: A description of the host system, and a description of the linguistic level the software system is intended to realize. The semantics of the linguistic level of the host system must be known before the components of outer software layers can have exact representations. Of course, the objectives of the system must be known before final designs of all of its components can be specified.

4. FUNCTION, CORRECTNESS, PERFORMANCE AND RELIABILITY

The designer of a software system wishes to achieve certain goals. The goals are expressed in terms of four kinds of properties desired of the completed software system: function, correctness, performance, and reliability. Let us consider the state-of-the-art in each of these four aspects of software systems and the directions in which further development of principle is needed.

4.1 FUNCTION

The function of a software system is the correspondence desired of output with input. Input is all information absorbed by the software system from outside the host system; output is all information delivered outside the host system. Information held by a software system between interactions with the outside is covered by this view, since such information either is the result of processing information received as input, or should be considered part of the software system, its effect then being incorporated in the mapping of inputs to outputs.

In the case of application software, the function of a software system depends on what one takes as the host system. For example, the data base for an application may be internal if the host system provides a data management facility, or it may be external if the data base is on a set of tapes not part of the host system.

In the case of system programs, the function of a collection of system programs is to implement a specified linguistic level. A linguistic level is adequately defined only by a model of a class of system states, and a state-transition function which, together, give the equivalent of a formal interpreter for the level.

There is a rapidly growing body of formal knowledge applicable to many aspects of the representation of programs and systems. Some of this material is listed below:

1. Semantic models for programming languages.
   - the lambda calculus [2]
   - the contour model [3, 4]
   - Vienna definition method [5, 6]
   - program schemas [7, 8]
2. Concepts relating to interacting concurrent activities
   - Petri nets [9]
   - processes, semaphores, determinacy [10]
   - modularity [11]
3. Fundamentals of classes of algorithms
   - numerical methods
   - symbolic algorithms (e.g. sorting, theorem proving)
   - parsing methods

Although the theoretical foundation for programs and systems is fast developing, there is as yet no generally accepted representation scheme that has a precisely known semantics and is sufficiently general to meet the descriptive needs of software system designers. Areas in which the theoretical development has not yet provided an accepted synthesis of concepts are:

1. Representation of concurrent activities and their interaction.
2. The sharing of procedures and data among computations.
3. Representation of data structures which change in content and extent during computation.
4. The notions of ownership, protection, and monitoring.

[illegible]

The consequence of this state of affairs is that designers of computer systems adopt different sets of primitive data types and operations as the basis for the design of the inner layers of hardware and software. Then, in realizing a standardized linguistic level such as a Fortran programming system the system designer employs these primitives to implement the standardized aspects of the language. Nevertheless, the implementer is usually forced to implement extension of the language so application programmers may make use of unstandardized linguistic features of the host. Since the primitives in terms of which these extensions are defined are different for different computer systems, the extensions are unlikely to be compatible, and portability of the application software is lost.

This discussion underscores the need for better understanding of the semantic issues listed above.

Suppose a computer system is developed as a hierarchy of several linguistic levels. Then the data types and primitive operation used at each linguistic level are restricted to those implemented at deeper levels. Often a single language (a system programming language) is advocated for representing software components at all levels within the system. In this case, either the language can include only the linguistic features implemented at the innermost level (the hardware), or restrictions must be placed on use of linguistic features depending on the level for which software is being written. Certain essential hardware features such as interrupt mechanisms, processor faults, and protection features are not usually incorporated as linguistic features of the system programming language, and recourse must be made to machine language procedures. In this way, the system programming language is extended to encompass the primitives required to implement its higher level features, and linguistic features of the computer system that are not directly encompassed by the system programming language.

Thus a system programming language provides primarily a syntactic structure permitting easy use of linguistic features common to all linguistic levels at which it is used. The degree to which a system programming language aids in simplifying the design and programming of system software depends critically on the generality of the set of linguistic features common to all software levels.

4.2 CORRECTNESS

Correctness of a software system means correctness of its description with respect to the objective of the software system as specified by the semantic description of the linguistic level it defines. Regardless of the approach adopted to favor correctness of a software system, it is always the responsibility of the designer of the system or system component to convince himself of the correctness of _some_ description of the system or component. One would like this description to be as simple as possible, for example, a simple relation of output to input.

Two approaches to the correctness of systems have been suggested:

1. Structured programming [12]: The use of a programming style that makes the correctness of a program self-evident to the author.

Greater use of structured programming is limited by the need for linguistic features not found in established programming languages. Use of structured programming may be encouraged by use of languages that disallow troublesome linguistic features such as _goto_ statements and side effects.

2. Proof of correctness [13]: To prove correctness of a software system or component, one establishes by logical deduction that some description of the system or component asserted to be correct by the designer is equivalent to the description of the system or component expressed at the host level.

In the case that the host level description is the result of automatically translating the designer's description, proving the correctness of the translator suffices. In other cases mechanically generated proofs or man-machine proof generating systems are required for this approach to be effective, and the semantics of the host language must be correctly axiomatized for the proof generator. This approach is beginning to be used experimentally. Although it is questionable whether establishing correctness by proof will become a practical technique, the research is yielding useful knowledge for improving the design of programs and languages.

4.3 PERFORMANCE

Performance of a software system is the effectiveness with which resources of the host system are utilized toward meeting the objective of the software system.

The demands on a contemporary software system usually cannot be modelled exactly, and statistical characterizations must be employed. The theoretical foundation for performance studies is Markov processes and queuing models, for these models of stochastic service systems are amenable to analysis. In software systems where the demands can be reasonably well determined by observation, for example, in real-time transaction systems, statistical analysis has provided valuable predictions of performance to system designers.

On the other hand, performance analysis has so far failed to provide adequate methods for predicting the performance of software systems where the applications to be implemented at the new linguistic level are unknown. This state of affairs is due to two difficulties, both stemming from the lack of generally accepted representation schemes for software. One difficulty is the absence of a satisfactory model of resource usage for application programs represented at the new linguistic level. For each design of a software system, a new model of program behavior has to be formulated and validated before it can be used to extrapolate performance data. These models have not been useful for predicting performance of a tentative system design. The other difficulty is that the software system itself is not represented in a generally accepted notation, and no standard techniques of performance analysis are available for direct application to the description of software systems.

The main point of these remarks is that our ability to analyze and predict performance of software systems is limited by the inadequacies of available description schemes rather than by the inadequacy of statistical methods. After all, approximate answers to performance questions are often satisfactory, but there is no such thing as a satisfactory approximate description of function.

4.4 RELIABILITY

Reliability is the ability of a software system to perform its function correctly in spite of failures of computer system components. By failure of a component we mean a temporary or permanent change in its characteristics that alters its function. Software does not fail. What is often referred to as "software failure" is a matter of correctness.

Nevertheless, one must recognize the high likelihood of incorrect software being present in a complex software system. The design of a system as a set of minimally interacting modules using principles of structured programming can limit effects of software bugs to the modules and data structures that depend on correctness of the module in error. The possibility of realizing practical systems constructed according to this principle depends on new fundamental knowledge of structured programming and modular systems.

If a software system has no hardware components, then component failures can only occur within the hardware components of the host computer system. In the ideal host system, failures of its hardware would not be observable at the linguistic level defined. Some current work [14] on fault-tolerant and self-testing and repair computer architecture is directed toward realizing this ideal, but is still far from solving the problem in the context of general purpose computer systems. Most reported work on reliability is concerned with the detection of failures and does not attempt to cope with the loss of information that inevitably accompanies hardware failure. We need concepts of computer organization that will permit the construction of computer systems in which single internal failures do not produce observable effects.

Since the ideal host system is not now available, some hardware failures will affect operation of software systems implemented at the linguistic level of the host. Then a description of the host system is not complete without a specification of the possible modes of failure, and the resulting effects observed at the host linguistic level. A software system to be implemented on such a host is not completely described unless the action to be taken for each failure mode of the host is specified.

At present we must be satisfied with software systems even if they occasionally fail with irrecoverable loss of information. For it is not known how to construct an infallible software system using a fallible computer system as host. Although there are systems (such as the American Airlines Sabre system and the Bell System's Electronic Switching System No. 1 ESS) that come close to providing complete protection against all single failures, the techniques used do not generalize easily to computer systems intended for general application.

5. SOFTWARE PROJECTS

Large projects for the design and construction of software systems are notorious for their delays in meeting specified objectives. A large project is one in which two or more levels of management are required, and hence the key personnel of the project are not in continuous communication with one another. In a large project it is necessary to divide the work to be done into units for assignment to project teams. Any unit of work which in itself amounts to a large project must be further subdivided.

The best division of work into units is the division that minimizes the interaction between units. Two kinds of structure may be used as a basis for the subdivision of work: hierarchy and modularity.

Suppose a project team is assigned the construction of some module of a software system. The team's task is completely defined by a precise specification of:

1. The function of the module.
2. The linguistic level of the host system.
3. The performance required of the module.
4. The performance capability of the host.

In practice this information is at best only partially known by a project team at the time it is expected to begin work. It is often still unknown even at the time the team is expected to have a usable version of the module ready for integration with other system components.

Clearly, the most crucial information required by a project team is a precise definition of the linguistic level of the host system: for it is impossible for the team to produce a correct description of any part of the module unless the semantics of the host system are known.

Iteration of design is frequently found to be necessary in large software projects. Iteration occurs when it is found that decisions already made prevent realization of overall system objectives. The most serious design iteration is where more than one linguistic level is affected, as the description of all modules in outer software layers may be invalidated by a change to a host system. The need for iteration arises in several ways: Sometimes it is discovered that certain linguistic features needed to implement a software system are impossible to realize in terms of the primitive constructs of the host level. Then the semantics of the host level must be revised to meet the need. In other cases, it is found that the performance objectives of a software system cannot be achieved without altering the specification of host level function.

These observations bring out the importance of having a precise specification of the host system before beginning construction of components of a software system. For each additional layer included in a software system, either the project must be extended to allow time for the precise formulation of the new linguistic level, or work on several levels must overlap, raising the risk that design iteration will be required. The need to implement several linguistic levels within one project would be circumvented if a host computer system were available that realized a complete and satisfactory linguistic level for the objectives of the project.

These arguments reinforce the need for better understanding of fundamental linguistic constructs for building software systems and the development of corresponding principles of computer system architecture. When this understanding has been gained, perhaps there will no longer be any need for large software projects.

6. ACKNOWLEDGEMENT

The author wishes to express his thanks to Professor Jerome Saltzer, whose incisive comments on an early draft have been valuable in the preparation of these notes.

7. REFERENCES

1. J. G. Kemeny and T. E. Kurtz, BASIC Programming. John Wiley and Sons, Inc., New York 1967.
2. P. J. Landin, A correspondence between ALGOL 60 and Church's lambda-notation, Part I: Comm. of the ACM, Vol. 8, No. 2 (February 1965), pp 89-101.
   Part II: Comm. of the ACM, Vol. 8, No. 3 (March 1965), pp 158-169.
3. J. B. Johnston, The contour model of block structured processes. Proceedings of a Symposium on Data Structures in Programming Languages, Sigplan Notices, Vol. 6, No. 2 (February 1971), pp 55-82.
4. D. M. Berry, Block structure: retention or deletion? Proceedings of the 3rd Annual ACM Symposium on Theory of Computing, May 1971, pp 86-100.

5. P. Lucas and K. Walk, On the formal description of PL/I. Annual Review in Automatic Programming, Vol. 6, Part 3, Pergamon Press, 1969.

6. P. Lucas, P. Lauer, and H. Stigleitner, Method and Notation for the Formal Definition of Programming Languages. Technical Report TR 25.087, IBM Laboratory Vienna, June 1968.

7. M. S. Paterson, Decision problems in computational models. Proceedings of an ACM Conference on Proving Assertions About Programs, SIGPLAN Notices, Vol. 7, No. 1 (January 1972), pp 74-82.

8. A. P. Ershov, Survey paper on program schemata, presented at the IFIP Congress, Ljubljana, 1971.

9. A. Holt, F. Commoner, S. Even, and A. Pnueli, Marked directed graphs. J. of Computer and System Sciences, Vol. 5, No. , (1971), pp 511-523.

10. E. W. Dijkstra, Co-operating sequential processes. Programming Languages, F. Genuys, Ed., Academic Press, New York 1968. [First published as Report EWD 123, Department of Mathematics, Technological University, Eindhoven, The Netherlands, 1965.]

11. S. S. Patil, Closure properties of interconnections of determinate systems. Record of the Project MAC Conference on Concurrent Systems and Parallel Computation, ACM, New York 1970, pp 107-116.

12. E. W. Dujkstra, A constructive approach to the problem of program correctness. BIT (Nordisk Tidskrift for Informations-behandling), Vol. 8, No. 3, 1968, pp 174-186.

13. Z. Manna and R. J. Waldinger, Toward automatic program synthesis, Comm. of the ACM, Vol. 14, No. 3 (March 1971), pp 151-165.

14. A. Avizienis, G. C. Gilley, F. P. Mathur, D. A. Rennels, J. A. Rohr, and D. K. Rubin, The STAR (Self-Testing and Repairing) computer: An investigation of the theory and practice of fault-tolerant computer design. IEEE Trans. on Computers, Vol. C-20, No. 11 (November 1971), pp 1312-1321.