



Software Structures: A Careful Look

David Lorge Parnas

IF ALL SOFTWARE experts agree on anything, it is that software shouldn't be a monolith (a large system that is, for all practical purposes, indivisible). In the half century since Edsger Dijkstra published his groundbreaking paper, "The Structure of the 'THE'-Multiprogramming System,"¹ it has become clear that the ability to design a software system's structure is at least as important as the ability to design efficient algorithms or to write code in a particular programming language.

Although the word "structure" appeared in the paper's title and was used seven more times, Dijkstra never defined the term. Closer examination revealed that he was discussing at least three distinct structures.² His failure to define "structure," or to clearly distinguish the structures that were important in his software, has led many to confuse those structures. This article is intended to clarify what those structures are, their differences, and each one's importance. They are as important today as they were in 1968.

What Is a Structure?

A software system's structure is a division of that system into a set of parts and the relations between those parts. Confusion often arises because of the many types of parts

and the many important relations between them.

What Parts Can We Discuss in Software?

The software world isn't known for using consistent terminology—quite the contrary. This section defines four terms used in this article. Other authors explicitly or implicitly use other definitions. To understand this article, it is important to keep these definitions in mind. A comparison of the concepts follows the definitions.

Program

Definition: A *program* is a sequence of computer instructions that can be invoked (executed) by a computer.

Discussion: An executing program will cause changes to the state of the machine. The changes might either continue for a finite amount of time and then terminate or continue forever.

Module

Definition: A *module* is a set of programs together with the data objects they use, whose design and subsequent maintenance (revision) are intended to be assigned to a developer or group of developers.

Discussion: Several modules might be assigned to the same developers. Over time, the team assigned

the responsibility for a module will change. The division into modules is most significant when the project has been assigned to a team rather than an individual. For further discussion of the meaning of "module" and the criteria to use when designing modules, see "The Modular Structure of Complex Systems."³

Component

Definition: A *component* is a collection of programs and data objects that is intended to be distributed, sold, installed, or replaced as a unit.

Discussion: The software product's purveyor determines the product's division into components. The purveyor might do this as part of a marketing strategy, for ease of installation, or to simplify management of the product line.

Process

Definition: A *process* is a sequence of events (state changes) in an executing computer system.

Discussion: A process is usually the result of executing a program, which we call the *process controller*. That program may be controlling other processes at the same time.

The adjective "sequential" is often inserted before "process" because

no two events in a process can occur simultaneously.

Processes are units for the scheduler. The scheduler assigns a processor to a process, thereby allowing the sequence of events to be extended. The maximum number of processes that can proceed simultaneously is the number of processors. Sometimes, the number of active processes is smaller than the number of processors, and some processors are idle.

In principle, processes should be designed so that no two events in a process could occur simultaneously without causing errors. If that isn't done, the scheduler will be unnecessarily restricted, and some processors might be unnecessarily idle. In practice, because of the cost of creating and managing processes, events might be included in a single process even though they could safely occur simultaneously.

Contrasting These Concepts

People sometimes use these terms without clearly distinguishing them. This section contrasts the concepts.

Program versus module. A program is part of one module but might invoke programs from other modules. In other words, programs written by one programmer often invoke programs written by other programmers. Usually, a module comprises more than one program.³

Component versus module. A component might be part of a module, a whole module, or a collection of programs from several modules. There is no reason to assume either that all parts of a component must be developed or maintained by the same programmers or that all programs in a module must be included in a component.

Program versus process. One program will often control many processes. Consequently, it is important to distinguish between a process and the program that controls it. The program controlling a process might invoke programs from many modules, and a module might perform its work by controlling several processes.

The bottom line. These concepts are distinct; no understanding of software structure can be correct if they are confused.

Four Important Structures

This section describes the parts and relation for four important structures. Others are discussed in "On a 'Buzzword': Hierarchical Structure."²

Program uses Program

In the *uses* structure, the parts are programs, and the defining relation, *uses*, is defined by the following:

Given a program P with specification S_p and another program Q with specification S_q , we say that P *uses* Q if P can't satisfy S_p unless Q is present and satisfies S_q .

Module part-of Module

If a module (work assignment) is considered too large, it can be subdivided into smaller modules. This defines a *part-of* relation between the modules. Note that the fact that P is part of M and invokes Q doesn't imply that Q is part of M . Q might have been written by another group.³

Program part-of Program

Dijkstra (among others) introduced *programming by stepwise refinement*. In this process, the programmer begins with a short program in

which major steps in the processing are represented by only a brief description of their function or simply a name. Each program is then *refined*—that is, replaced by a more detailed program. That program might be further refined in the same way until all its parts have been refined to programs in the actual programming language. This process produces a program with a clear structure (often called a structured program) that is easily parsed, documented, and analyzed.⁴

A program might use a program that isn't part of it. A program might be used by a program that is part of another module.

Process gives-work-to Process.

In many systems, processes give work to other processes. For example, one process might produce output that must be printed and send it to another process that controls the printer. In the *gives-work-to* structure, the parts are processes, and the defining relation is *gives-work-to*.

The Significance of Each Structure

These structures were important in the THE operating system.

The *uses* structure determined what subsets of the system were usable. Because there were no loops in the graph of the *uses* relation, THE programs could be arranged in levels, with the lowest level being programs that didn't use any other programs and the i th level consisting of programs that used programs on level $i - 1$ or lower. As a result of the hierarchical structure, construction and testing could be done bottom-up. That is, the programs that used no other programs could be finished and tested before those that used them. This could continue level by

level until the highest-level programs had been tested.

The modules in THE were information-hiding modules³ and introduced abstractions such as

- processes that appeared to proceed in parallel,
- a virtual memory for each process that was larger than the actual main memory,
- a private console for each process, and
- FIFO queues (transput streams) as a mechanism for passing data between processes.

Dijkstra's computer didn't have hardware that supported virtual

called *structured programming*. Dijkstra, in private discussions, often stated that structured programming was the only way to write reliable programs.

When Dijkstra wrote about the "system hierarchy," he didn't make clear which of these structures he was discussing. In THE, the *uses* structure and the module structure coincided. The programs in the module that created the sequential-process abstraction used no other programs and were all on level 0 of the structure defined by the *uses* relation. The programs in the module that created the virtual-memory abstraction took advantage of the sequential processes when managing

that he could have split the virtual-memory module, putting the programs that translated virtual addresses to physical addresses at the lowest level. This would have allowed the programs that implemented the sequential-process abstraction to use virtual addresses. The remainder of the virtual-memory module's programs, which managed memory allocation, could then have used processes. In today's computers, the address translation is done by hardware, which is actually the lowest level in the *uses* structure. The programs that manage physical memory to implement virtual memory are at higher levels.

If Dijkstra had rebuilt his system in this way, the *uses* hierarchy would have had more levels than there were modules. Consequently, it would be wrong to talk about levels of abstraction. There would still be levels in the *uses* structure—one more than in the published version. There would still be the same modules implementing the same abstractions. However, the two structures would no longer coincide. Other examples of module structures in which the programs aren't all at the same level of the *uses* hierarchy are explained in "Designing Software for Ease of Extension and Contraction."⁶

Each module abstracts from something different; there's no "more abstract than" relation between the modules. In general, the programs that implement an abstraction can be placed in many levels in a system's *uses* hierarchy. When designers insist on levels of abstraction, it can lead to poor performance or duplication of certain functions.

Design and Document Structures before Coding

Dijkstra's papers and presentations made it clear that he and his team

Designing and documenting software structure is as demanding as coding.

memory. All translation from virtual to physical addresses was done by programs in the module that implemented the virtual address space. In the original THE operating system, all those programs were on the same level of the *uses* hierarchy.

The *gives-work-to* structure was designed to help prevent deadlocks, as explained in the PhD thesis of Nico Habermann,⁵ a member of Dijkstra's team. A badly designed *gives-work-to* structure can result in system deadlocks. There's no reason why this structure should coincide with any other.

We must assume that the programs were structured as a result of stepwise refinement, which Dijkstra

page transfers. The transput stream module's programs could use the virtual-memory abstraction.

However, Dijkstra later realized that the two structures needn't coincide. He reported that he would have liked the sequential-process module to be able to use virtual addresses, but he also wanted the virtual-memory module to be able to use the process abstraction. When developing THE, he believed that the two structures should coincide, so both wouldn't be possible. He chose to write the process-creating module using physical addresses so that the virtual-memory module could use processes.

Much later, in a presentation and an informal note, he explained

designed the structures before any code was written. The structure guided the coders; the result was a design that was “cleaner” than other systems of that time.

Dijkstra’s team was small, highly motivated, and very talented. Dijkstra, who described himself as the team’s captain, was very hands-on. They didn’t create precise documentation of the structure. With a larger team, one that was managed rather than “captained,” the lack of documentation would have led to miscommunication. That would have lengthened the development time and might have introduced errors.

The lack of documentation became evident after Dijkstra’s team dispersed. The system was their legacy and was used for some time after they left. One member of the original team frequently received phone requests for help when a problem occurred. In that team member’s words, “The structure was clean and simple, but it existed only in our minds.” The lesson is clear: structure is vital, but unless you plan to discard the software when its authors move on, it must be documented.⁷

If all software experts agree on anything, it is that software shouldn’t be a monolith. However, merely dividing the software into parts and discussing some of the relations between them is not enough to solve the real problem. Many distinct structures determine the software’s cost and quality. Each structure must be carefully designed and precisely documented.


Designing and documenting software structure is as demanding as coding. Small details matter,



ABOUT THE AUTHOR



DAVID LORGE PARNAS is professor emeritus at McMaster University and the University of Limerick. He is the president of Middle Road Software. Contact him at parnas@mcmaster.ca.

clear design principles are needed, and precise interface specification is essential. 

S. Nanz, ed., Springer, 2010, pp. 125–148; doi:10.1007/978-3-642-15187-3_8.

References

1. E.W. Dijkstra, “The Structure of the ‘THE’-Multiprogramming System,” *Comm. ACM*, vol. 11, no. 5, 1968, pp. 341–346; <http://dx.doi.org/10.1145/363095.363143>.
2. D.L. Parnas, “On a ‘Buzzword’: Hierarchical Structure,” *Proc. 1974 IFIP Congress*, North Holland, 1974, pp. 336–339.
3. D.L. Parnas, P.C. Clements, and D.M. Weiss, “The Modular Structure of Complex Systems,” *IEEE Trans. Software Eng.*, vol. 11, no. 3, 1985, pp. 259–266.
4. D.L. Parnas, J. Madey, and M. Iglewski, “Precise Documentation of Well-Structured Programs,” *IEEE Trans. Software Eng.*, vol. 20, no. 12, 1994, pp. 948–976.
5. A.N. Habermann, “On the Harmonious Cooperation of Abstract Machines,” PhD thesis, Dept. of Mathematics, Technological Univ. Eindhoven, 1967.
6. D.L. Parnas, “Designing Software for Ease of Extension and Contraction,” *IEEE Trans. Software Eng.*, vol. 5, no. 2, 1979, pp. 128–138.
7. D.L. Parnas, “Precise Documentation: The Key to Better Software,” *The Future of Software Engineering*,

myCS Read your subscriptions through the myCS publications portal at <http://mycs.computer.org>

IEEE **SECURITY & PRIVACY**



@securityprivacy