Engineering design: Its importance for software

Ann Miller

For software development, analysis and design documents are far more than "just for the record"

Many students just graduating from a university program are skeptical about the need for a software design methodology. One reason for this doubt is the difference in project size. Generally, most students have written small programs on an individual basis. However, most industrial software projects are considerably more complex and require a team effort. Even "small" industrial projects require a team of engineers, perhaps a mix of hardware and software engineers, and extend over several months.

For a very small programming task, it is possible to write a logical, correct, and readable program without performing the up-front analysis and design procedures discussed here. An individual can understand the architecture and function of a 100-200 line program by reading the header and comments within the code. However, imaging trying to read your way through a code listing that requires six feet of bookshelf to store. It's easy to become lost in the details. Some other form of description is necessary.

A diagrammatic representation is especially helpful—this is merely a fancy way of saying that a picture is worth a thousand words. So, one advantage to a design methodology is that it provides high-level information about the software, and it does so with graphical aids.

Software reuse

The cost of developing software has skyrocketed. Further, the time to develop software has often caused systems delivery to be late. In an attempt to decrease cost and to reduce development time, many individuals are exploring software reuse. In theory, it is a wonderful concept: imagine having whole libraries of functions, software packages, device drivers, etc., that can be incorporated into new programs. So, with all of the obvious advantages, why isn't software reuse a reality? The problem is that software reuse will not happen easily unless there is the underlying design reuse. The reason is simple enough: if you are developing a new software product, you need to know how a reused package will fit into your system. Thus, you need to know not only *what* function it performs, but *how* it is invoked, *what* parameters must be passed, and *what* their structures are. In other words, you need design information. Thus, design reuse must be the basis for software reuse.

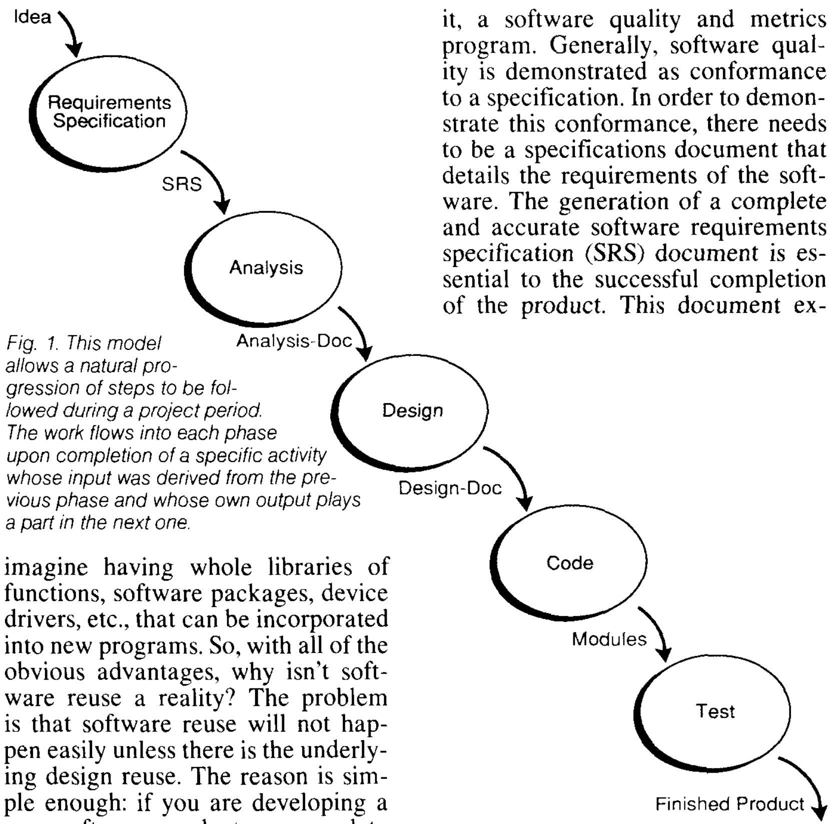
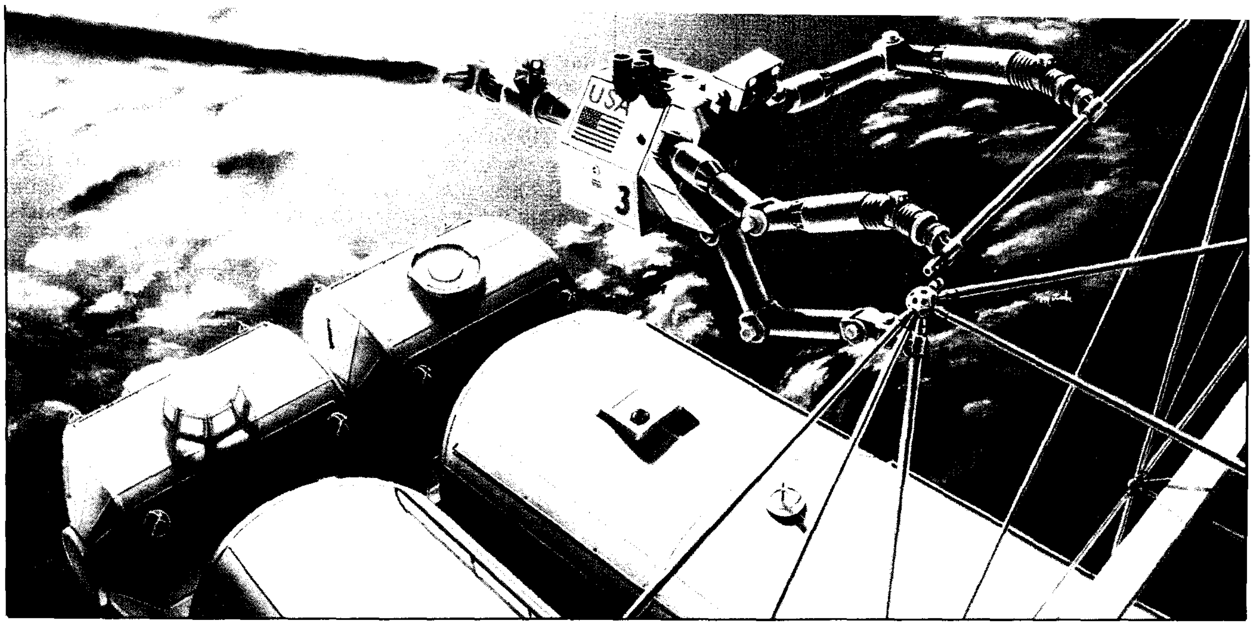


Fig. 1. This model allows a natural progression of steps to be followed during a project period. The work flows into each phase upon completion of a specific activity whose input was derived from the previous phase and whose own output plays a part in the next one.

Software quality and reliability

Another hot topic in the software community is software quality and reliability. While this is not a "new" topic, it has become increasingly important. Nearly every company involved in the production of software has a "quality slogan" and with it, a software quality and metrics program. Generally, software quality is demonstrated as conformance to a specification. In order to demonstrate this conformance, there needs to be a specifications document that details the requirements of the software. The generation of a complete and accurate software requirements specification (SRS) document is essential to the successful completion of the product. This document explains what is to be accomplished. And it is against this document that the acceptance test can judge if the requirements have been met.

What is involved in writing the SRS and how does one demonstrate that the specifications have been met? First, the specifications need to meet several criteria; they should be complete, consistent, accurate, unambiguous, and testable. While most of these characteristics are obvious, the last one deserves further discussion. Generally speaking, testable requirements are those written in as quantitative a manner as possible. For example, rather than writing a requirement such as, "good response



Illustrated here is NASA's flight telerobotic servicer assembling part of the U.S. space station Freedom in orbit. This robot will aid astronauts in this project during the mid-1990s; afterwards, it will service the station and other spacecraft on its own. The software requirements alone for the advanced robotics and artificial intelligence will be complex. Martin Marietta's Space System is currently analyzing the servicer for NASA's Goddard Space Flight Center.

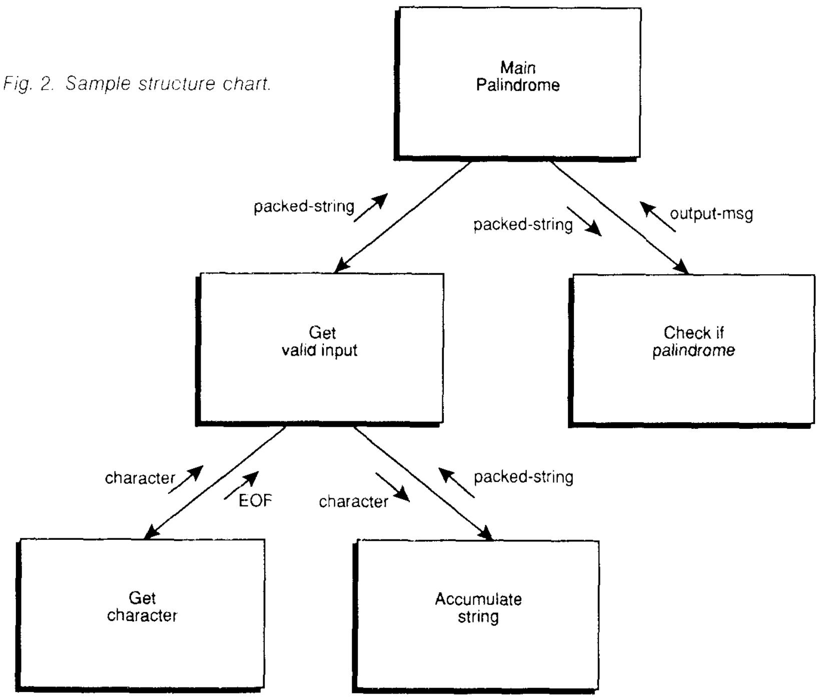
time," for a terminal interface, a better phrasing would be, "sub-second response time," since this statement can be measured.

More and more, formal techniques and specification languages are being proposed in order to circumvent the problems that arise from using a natural language, such as English, with its inherent ambiguities and imprecise terminology. In fact, if the specifications could be written in a language that was machine executable, then once everyone agreed on the specifications, the software would be written! In fact, research into *executable specifications* is an important on-going activity.

At present, however, most specifications are written in a natural language and the SRS is a textual description. So, how do we demonstrate that the resulting software meets the specifications? Most industrial software development projects utilize a software life cycle as shown in Figure 1. This cycle is called the waterfall model, because each phase is supposed to flow naturally from its predecessor and into its successor stage. In this model, software progresses from a requirements specification to an analysis document to a design document to independent code modules to an integrated software package. The defining characteristics of a phase are that: (i) there is a specific activity associated with that phase, (ii) there is an identified input to the phase, and (iii) there is an identified output from the phase. In fact, the output of each of these phases is part of the complete software, even though only output of the latter stages can be executed. One method to show conformance to the SRS is to trace what happens to the specific requirements as the phases are completed. This is known as *traceability of requirements*.

The result of the design phase is a complete description of the software modules to be produced, their interfaces to each other, and their relationship in the calling sequence. There are a variety of methodologies, each with their own notation, that can be used to produce this architectural and interface information. One approach is a structure chart, a small sample of which is shown in Figure 2.

Fig. 2. Sample structure chart.



Reduction of communication problems

One advantage of a structure chart is that we can "cut it up" and assign tasks to individual team members during the next phase. Let's suppose that you and I are two of three software engineers who will be developing a new software project. The division of work not only includes the assignment of subtasks to individuals, but also the interfaces between subtasks. This is, if you are assigned the time-keeping functions of a system and I am writing the clock interrupt service routines, you need to understand how I am passing information and where I will place that information. Thus, you and I as developers need to communicate during the development process. If we consider the communication paths necessary for our team of three engineers, we have a simple triangle as illustrated in Figure 3a. However, if the development team is doubled and now consists of six engineers, the number of possible communication paths is more than double; see Figure 3b. Imagine the potential communication paths in a team of 100 or 300 engineers. Teams of this size do exist for large-scale software development, such as an operating system or a switch for telephone systems. Admittedly, not every engineer assigned to a subtask for a large-scale project needs to communicate with every other engineer, but the number of channels is still quite large. A reduction in communications can occur if the documents from one phase provide a thorough specification of the tasks to be performed in the next phase.

Moreover, if each box in the structure chart has an accompanying description (called a module specification), then the programmer for that module knows what functionality it needs. Further, the interfaces between each module are described; these are the data and control items indicated by arrows. A textual description (in a "dictionary") describes each parameter, including its form, its range, and its function. Thus, the structure chart and parameter dictionary, which are the output of the design phase, can be assigned in pieces to the team of programmers to perform the next phase, which is the coding stage.

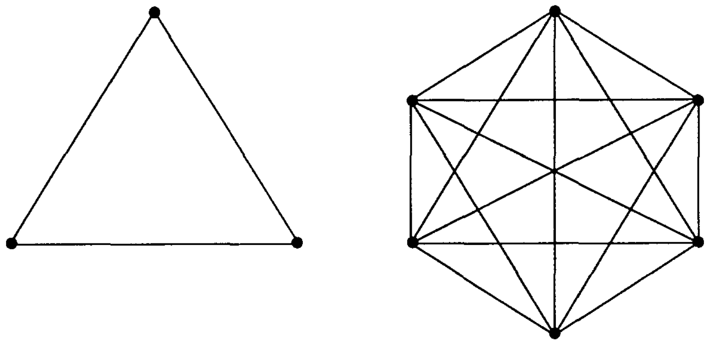


Fig. 3. (a) Communication paths for a team of 3. (b) Communication paths for a team of 6.

Revisions

The basic disadvantage of the waterfall model is that, in reality, things don't proceed as simply as implied in Figure 1. During a design review, a major flaw can be uncovered that requires going back and correcting portions of the analysis document. However, the model, with iterations as just mentioned, has been used successfully in industry and government. So, with regard to revisions, it is easier and cheaper to change the relationships between modules on a sketch or figure than to change the code within those modules.

Software testing

Testing also benefits from engineering design. When most people think of software testing, they think of dynamic testing, that is, execution of code. While each unit test, systems integration test, and acceptance test are important, "testing" really goes on throughout the life cycle. Because of the nature of the waterfall model, there are natural test points, namely, the end of each phase. At these points, the document delivered from the phase is reviewed for accuracy, correctness, and completeness. Each of these reviews is a form of static testing.

Software maintenance

The analysis and design documents provide a complete description of the architecture of the software and the relationships between components (or modules) of the software. Without this information, it is difficult—if not impossible—to adequately maintain a software system. As changes are made to the existing code, it is important to understand the effect of the modification on the remainder of the system. The architectural and interface information from the early phases allow us to assess trade-offs in modifications and to determine the potential side effects of these changes.

Software design methodologies

There are a variety of methodologies for software development. One of the oldest and most established is the structured analysis and design techniques originally proposed by Yourdon, Constantine, and others, which grew out of data processing applications of the 1950s. Over the years, it has been expanded and tailored for various applications, including real-time software. Some of the real-time extensions have been proposed by Ward and Mellor and by Gomaa. Data-structured design is another approach that is practiced. Lastly, a design methodology quickly gaining favor is object-oriented design.

Read more about it

- Gomaa, H., "A Software Design Method for Real Time Systems," *Communications of the ACM*, Vol. 29, No. 9, Sept. 1984, pp. 938-949.
- Mellor, S. J., and P. T. Ward, *Structured Development for Real-Time Systems*, Volumes I, II, and III, Yourdon Press, 1985, 1986.
- Pressman, R. S., *Software Engineering: A Practioner's Approach*, 2nd Ed., McGraw-Hill, 1987.

About the author

Dr. Ann Miller is an Associate Professor of Electrical and Computer Engineering at the University of New Mexico. Prior to joining UNM, she was a member of the technical staff in Corporate Research and Development at Motorola, Inc. Dr. Miller's research interests include real-time embedded software, software engineering design methodologies, and applications of artificial intelligence to software engineering and to computer-aided design. She is also an *IEEE Potentials* Associate Editor. □