# Requirements Engineering

Merlin Dorfman

Editor's Note: The following article is reprinted from the book *Software Requirements Engineering, Second Edition*, and is provided for readers who want to read a brief tutorial on requirements engineering. The views expressed in this article are the author's only and do not represent directly or imply any official position or view of the Software Engineering Institute or Carnegie Mellon University.



Requirements engineering is presented and discussed as a part of systems engineering and software systems engineering. The need for good requirements engineering, and the consequences of a lack of it, are most apparent in systems that are all or mostly software. Requirements Engineering approaches are closely tied to the life cycle or process model used. A distinction is made between requirements engineering at the system level and at lower levels, such as software elements. The fundamentals of requirements engineering are defined and presented: elicitation; decomposition and abstraction; allocation, flowdown, and traceability; interfaces; validation and verification. Requirements development approaches, tools, and methods, and their capabilities and limitations, are briefly discussed.

## Introduction

When the "Software Crisis"[1] was discovered and named in the 1960s, much effort was directed at finding the causes of the now-familiar syndrome of problems. The investigations determined that requirements deficiencies are among the most important contributors to the problem: "In nearly every software project which fails to meet performance and cost goals, requirements inadequacies play a major and expensive role in project failure." [2] Development of the requirements specification "in many cases seems trivial, but it is probably the part of the process which leads to more failures than any other." [3]

It was determined that the benefits of good requirements include:

- Agreement among developers, customers, and users on the job to be done and the acceptance criteria for the delivered system

- A sound basis for resource estimation (cost, personnel quantity and skills, equipment, and time)

- Improved system usability, maintainability, and other quality attributes

- The achievement of goals with minimum resources (less rework, fewer omissions and misunderstandings)

It was also observed that the value of good requirements, and the criticality of doing them well, increased dramatically with the size and complexity of the system being developed. Additionally, software-intensive systems seemed to have more inherent complexity, that is, were more difficult to understand, than systems that did not contain a great deal of software; thus these systems were more sensitive to the quality of their requirements.

The products of a good requirements analysis include not only definition, but proper documentation, of the functions, performance, internal and external interfaces, and quality attributes of the system under development, as well as any valid constraints on the system design or the development process.

As the value of good requirements became clear, the focus of investigation shifted to the requirements themselves: how should they be developed? How can developers know when a set of requirements is good? What standards, tools, and methods can help; do they exist, or must they be developed? These investigations are by no means complete: not only are new tools and methods appearing almost daily, but overall approaches to requirements, and how they fit into the system life cycle, are evolving rapidly. As a result, requirements engineering has been well established as a part of systems engineering. Requirements engineers perform requirements analysis and definition on specific projects as well as investigate in the abstract how requirements should be developed.

## Requirements engineering and the development life cycle

Many models exist for the system and/or software life cycle, the series of steps that a system goes through from first realization of need through construction, operation, and retirement.[4] (Boehm[5] provides a good overview of many existing models, and presents as well a risk-driven approach that includes many other models as subsets; Davis et al.[6] describe conditions under which various models might be used.) Almost all models include one or more phases with a name like “requirements analysis” or “user needs development.” Many models require generation of a document called, or serving the function of, a requirements specification. Even those that do not call for such a document,

for example Jackson System Development, have a product such as a diagram or diagrams that incorporate or express the user's needs and the development objectives.[7]

A few of the better-known life cycle models are briefly discussed in the following sections, and the way requirements engineering fits into them are presented.

## Baseline management

Among the most extensively used models are baseline management and the waterfall, on which baseline management is based.[8] (baseline management differs from the waterfall in that it specifically requires each life cycle phase to generate defined products, which must pass a review and be placed under configuration control before the next phase begins.) In these models, as shown in Figure 1, determination of requirements should be complete, or nearly so, before any implementation begins. baseline management provides a high degree of management visibility and control, has been found suitable for developments of very large size in which less complex methods often fail, and is required under many military standards and commercial contracts. This model, however, has been somewhat discredited, because when large complex systems are developed in practice it is usually impossible to develop an accurate set of requirements that will remain stable throughout the months or years of development that follow completion of the requirements. This essential and almost unavoidable difficulty of the waterfall and baseline management models had been noted for many years[9, 10] but was brought to the attention of the U.S. defense software community by a Defense Science Board report authored by F. Brooks.[11] Brooks pointed out that the user often did not know what the requirements actually were, and even if they could be determined at some point in time they were almost certain to change. To resolve this problem Brooks recommended an evolutionary model, as is discussed below. The approach advocated by Brooks provides the following advantages:

- The user is given some form of operational system to review (a prototype or early evolution), which provides better definition of the true needs than can be achieved by reading a draft specification. This approach avoids what Brooks identified as the unacceptable risk of building a system to the a priori requirements.

- Delivery of some operational capabilities early in the development process—as opposed to the delivery of everything after many months or years—permits incorporation of new requirements and of capabilities that did not exist or were not feasible at the start of development.

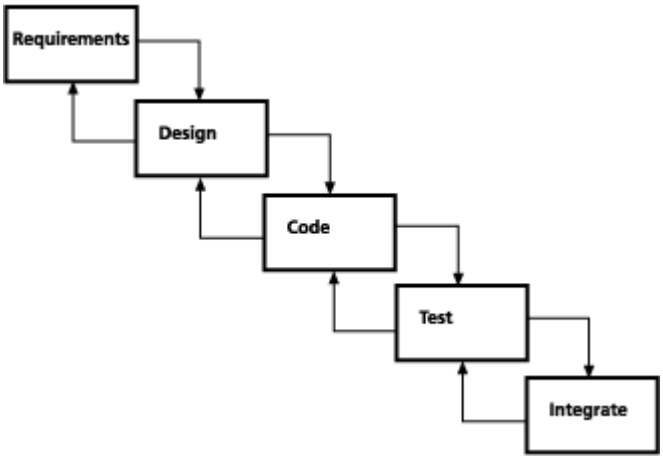


**Figure 1**: **The baseline management and waterfall models**

## Prototyping

The prototyping life cycle (Figure 2) is one approach to the use of an operational system to help determine requirements.[12] In this model, some system capability is built with minimum formality and control to be run for or by the user, so that requirements can be determined accurately. Several successive prototypes will usually be built. The amount of requirements analysis that precedes prototyping depends on the specifics of the problem. It is normally recommended that the prototype should be used only to help generate a valid set of requirements; after the requirements are available, they should be documented, and development should proceed as in the baseline management model. If this recommendation is followed, the prototyping portion of the life cycle may be considered as a tool or method supporting requirements analysis within the baseline management model.

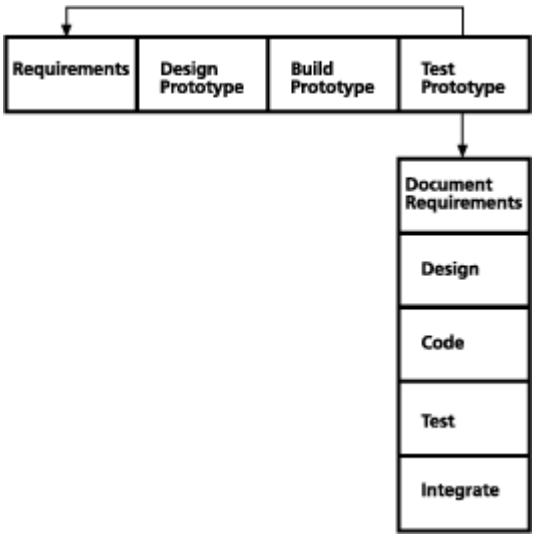


**Figure 2**: The Prototyping life cycle model

Many tools and methods are available to help support prototyping. The term *rapid prototyping* is associated with some of them, to distinguish them from other forms, such as development of high-risk hardware and software components, which is a slow, expensive process. Much of rapid prototyping is concentrated in two application areas: user interfaces and heavily transaction-oriented functions such as database operations. In these areas, a distinction can be made between prototyping tools and approaches that provide only "mockups" (simulate the system's response to user actions) and those that actually perform the operations requested by the user. In the latter category are the so-called fourth generation languages (4GLs),[13, 14] which provide methods of generating code for user operations included within the 4GL's capability. Such tools provide the option of retaining the prototype as (part of) the final system; considerations of execution time and efficiency of memory usage are weighed against the time and cost of building a system using the baseline management model and requirements determined from the rapid prototyping effort.

## Incremental development

The incremental development life cycle model calls for a unitary requirements analysis and specification effort, with requirements and capabilities allocated to a series of increments that are distinct but may overlap each other (Figure 3). In its original conception the requirements are assumed to be stable, as in the baseline management model, but in practice the requirements for later increments may be changed through technology advancement or experience with the early deliveries; hence this model may in effect be not very different from the evolutionary development model described next.

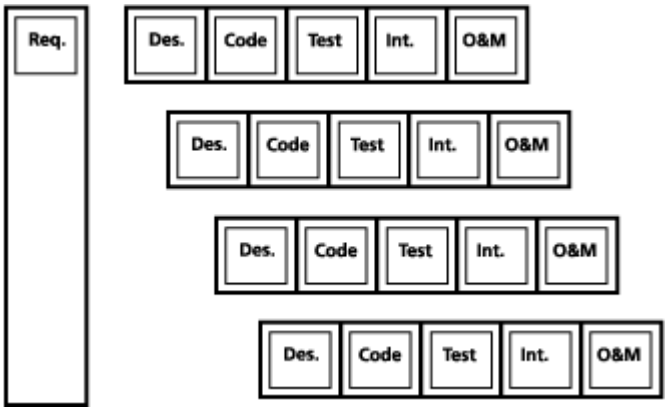


**Figure 3**: **The incremental development model**

## Evolutionary development

The evolutionary development life cycle calls for a series of development efforts, each of which leads to a delivered product to be used in the operational environment over an extended period of time (Figure 4). In contrast with the prototyping model, in which the purpose of each early product is only to help determine requirements, each delivery or evolution provides some needed operational capability. However, there is feedback from users of the operational systems that may affect requirements for later deliveries.

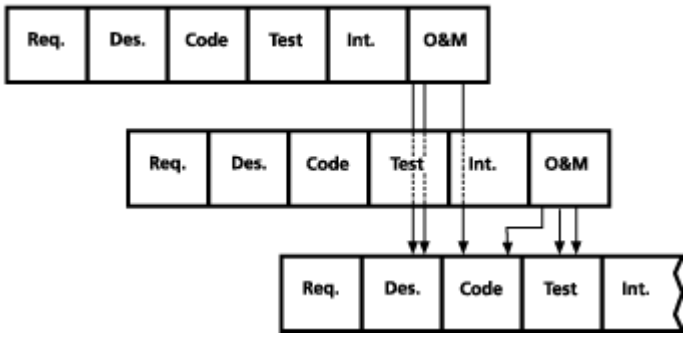


**Figure 4**: **The evolutionary development model**

Each delivery in this model represents a full development cycle, including requirements analysis. The deliveries may overlap, as shown in Figure 4, or one delivery may be completed before the next is begun. The product of each requirements analysis phase is an addition or improvement to the product(s) of the requirements analysis phase of the previous delivery. Similarly, the implementation portions of each delivery may add to, or upgrade, products of earlier deliveries. With this understanding, each delivery may be looked at as a small example of a baseline management life cycle, with a development process and time span small enough to minimize the problems discussed above.

## The spiral model

Boehm [5] describes the spiral model, an innovation that permits combinations of the conventional (baseline management), prototyping, and incremental models to be used for various portions of a development. It shifts the management emphasis from developmental products to risk, and explicitly calls for evaluations as to whether a project should be terminated. Figure 5 summarizes the spiral model.

The radial coordinate in Figure 5 represents total costs incurred to date. Each loop of the spiral, from the (–*x*) axis clockwise through 360 degrees, represents one phase of the development. A phase may be specification oriented, prototyping oriented, an evolutionary development step, or one of a number of other variants; the decision on which form to use (or whether to discontinue the project) is made at each crossing of the (–*x*) axis by evaluating objectives, constraints, alternatives, and status (particularly risk).

The spiral model thus makes explicit the idea that the form of a development cannot be precisely determined in advance of the development: the re-evaluation at the completion of each spiral allows for changes in user perceptions, results of prototypes or early versions, technology advances, risk determinations, and financial or other factors to affect the development from that point on.

Boehm has referred to the spiral model as a "process model generator": given a set of conditions, the spiral produces a more detailed development model.[15] For example, in the situation where requirements can be determined in advance and risk is low, the spiral will result in a baseline management approach. If requirements are less certain, other models such as the incremental or prototyping can be derived from the spiral. And, as mentioned above, re-evaluation after each spiral allows for changes in what had been (tentatively) concluded earlier in the development.

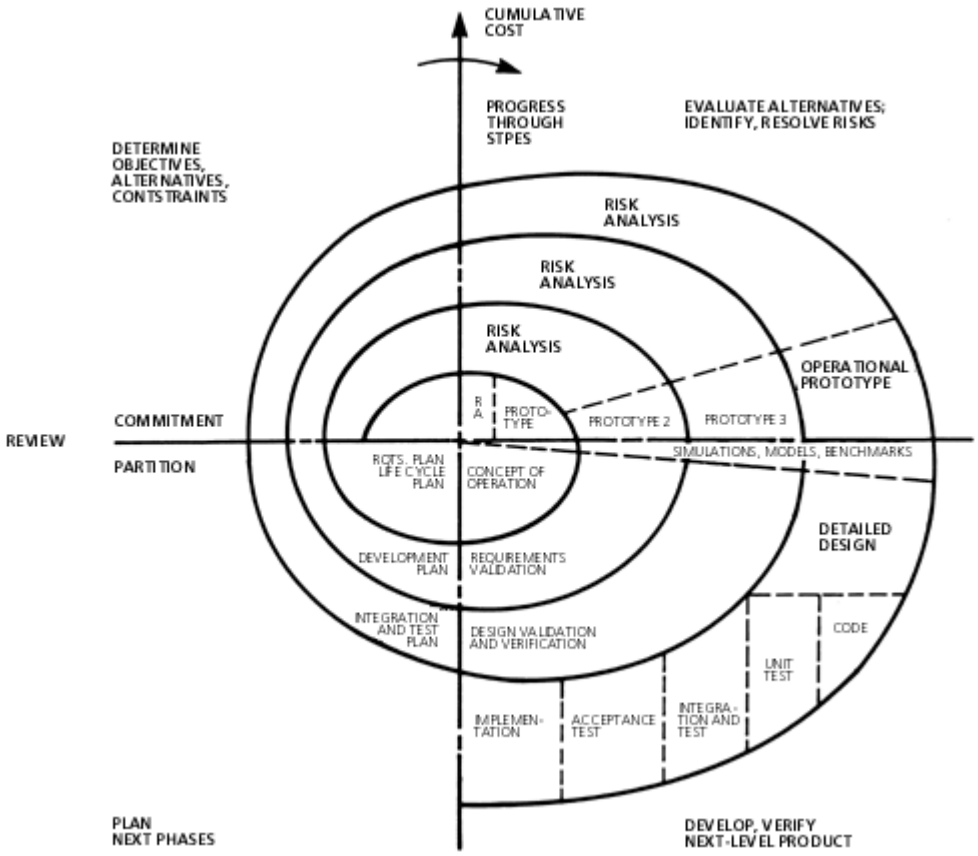


**Figure 5**: **The spiral model**

## System and software requirements

System and software requirements are often treated together because the tools and methods used to derive them, and the techniques of documenting them, are very similar. (It may be remarked that most of the tools and methods originated with software developers and were then found to be appropriate for system use as well.) However, some important differences between system and software requirements should be pointed out.

System requirements describe the behavior of the system as seen from the outside, for example, by the user. Although requirements documents and specifications cannot easily be read by users,[16] the system requirements serve at least as a partial communications vehicle to the more technically inclined users, or to a purchasing organization, as well as to analysts and designers who are concerned with the development of system elements or components.

Requirements for elements below the system level, whether they are for elements that are all hardware, all software, or composite (both hardware and software), are normally of minimal interest to users. These requirements serve to communicate with the developers, who need to know what is expected of the elements for which they are responsible, and who also need information about those elements with which they must interface.

This distinction is important for several reasons. First, like any communications vehicle, a requirements document should be written with its intended audience in mind. The degree to which nontechnical users must read and understand a requirements document is a factor in how it is written. Second, and perhaps most important, the skills and experience of the requirements developers must be considered. All too frequently, systems engineers with limited software knowledge are responsible for software-intensive systems. They not only write system-level requirements, which demands knowledge of what functions and performance a software-intensive system can be expected to meet; they also allocate functions and requirements to hardware and software. Thus the software developers are asked to develop designs to meet requirements that may be highly unrealistic or unfeasible. Software developers seem to be reluctant to get involved in systems engineering; one of the results is that the development of software elements often starts with poor requirements.

The U.S. Air Force Aeronautical Systems Center has recognized the importance of systems engineering to a software development by including in its software development capability evaluation (SDCE)[17] material about systems engineering capability and its interface with software engineering. The SDCE is an instrument used to help acquisition organizations determine whether bidders who have submitted proposals for a software contract are likely to be able to perform acceptably.

Section IV carries further the distinction between system and software requirements as the approach to generating requirements for all system elements is outlined.

## Fundamentals of requirements engineering

This section presents the overall framework within which requirements engineering takes place. Information about tools and methods is not presented here; at this point concern is focused on the sequence of events.

Several taxonomies have been proposed for requirements engineering. Prof. Alan Davis has proposed the following: [18]

- Elicitation
- Solution determination
- Specification
- Maintenance

Another is that requirements engineering consists of elicitation, analysis, specification, validation/verification, and management. The comparison with Davis's is straightforward—validation/verification is included in maintenance, and management is implicit in all four of Davis's activities.

A system of any but the smallest size will be decomposed into a hierarchy of elements. Starting with the lowest levels, the elements are integrated into larger-size elements at higher levels, back up to the full system (Figure 6). Several approaches are available for development of the hierarchy, but all produce definitions of elements at all levels of the hierarchy. In the functional approach (implied by the element names used in Figure 6), the elements represent the parts of the system that will meet particular system requirements or carry out particular system capabilities. In a physical decomposition, the elements will represent physical components of the system. In a data-driven approach, the components will contain different parts of the key data needed by the system, and most likely also the operations carried out on that data. In an object-oriented approach, the components will consist of *objects* that include not only physical components of the system but also the data and functions (operations) needed by those physical components.

After the lowest-level elements (*units* in Figure 6) are defined, they are separately developed and then integrated to form the next larger elements (*programs* in Figure 6). These elements are then integrated into larger-size elements at the next level, and so on until the entire system has been developed, integrated, and tested. Thus the life cycle models presented earlier can be seen to be oversimplified in that they do not account for the development and integration of the various elements that compose the system. This aspect can be considered to be outside the responsibility of the requirements engineer, but, as discussed below, it affects cost, feasibility, and other factors that bring the realities of implementation to the development of requirements, in contrast to the strict separation of "what" and "how" often postulated for system development.

The requirements engineer needs to be concerned with all the requirements work that takes place. As described above, this includes requirements for the entire system, for composite (hardware and software) elements at lower levels, and for all-hardware and all-software elements. It should be noted at this point that references to *the* "requirements specification" are meaningless for any but the smallest systems; there will be several or many requirements specifications.

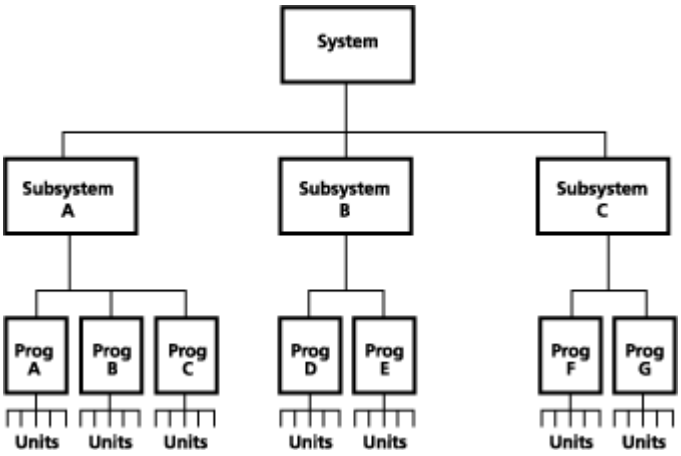


**Figure 6**: **Example system hierarchy**

## System requirements

Next is described the mechanics of filling out the system hierarchy with requirements.[19, 20] It should be pointed out that, although the hierarchy used as an example is functional, the process, and the traceability aspects that accompany it, are valid whether the decomposition is functional, physical, data driven, or object oriented.

Early in the development process, the system-level requirements are generated. A primary tool used in generating the system requirements is the Concept of Operations or ConOps document,[16] a document that is narrative in form and describes the environment and operation of the system to be built. An important part of the development of both the ConOps and the system requirements is the process of requirements elicitation, which is defined as working with the eventual users of the system under development to determine their needs.[21] Elicitation involves an understanding of psychological and sociological methods as well as system development and the application domain of the system to be built.

While the system requirements are being developed, requirements engineers and others begin to consider what elements should be defined in the hierarchy. By the time the system requirements are complete in draft form, a tentative definition of at least one and possibly two levels should be available. This definition will include names and general functions of the elements. Definition of the system hierarchy is often referred to as *partitioning*.

## Allocation

The next step is usually called *allocation*. Each system-level requirement is allocated to one or more elements at the next level; that is, it is determined which elements will participate in meeting the requirement. In performing the allocation, it will become apparent that (1) the system requirements need to be changed (additions, deletions, and corrections), and (2) the definitions of the elements are not correct. The allocation process therefore is iterative, leading eventually to a complete allocation of the system requirements, as shown in Figure 7. The table in Figure 7 shows which element or elements will meet each system requirement; all requirements must be allocated to at least one element at the next level. In this example, we have called the level below system the Subsystem level; in practice, the names are arbitrary, depending on the number of levels in the entire hierarchy and the conventions in use by the systems engineering organization. Figure 7 shows that the system-level requirement denoted as SYS001 is allocated to subsystems A and B, SYS002 is allocated to A and C, and so forth.

| SYSTEM REQUIREMENTS | SUBSYSTEM A | SUBSYSTEM B | SUBSYSTEM C |
|---|---|---|---|
| SYS 001 | X | X | |
| SYS 002 | X | | X |
| SYS 003 | | X | |
| SYS 004 | X | X | X |
| SYS 005 | | | X |
| SYS 006 | X | X | |
| SYS 007 | | | |

**Figure 7**: **Example of allocation of system requirements**

## Flowdown

The next step is referred to as *flowdown*. (The reader should be aware that this nomenclature is not universal.) Flowdown consists of writing requirements for the lower-level elements in response to the allocation. When a system requirement is allocated to a subsystem, the subsystem must have at least one requirement that responds to that allocation. Usually more than one requirement will be written. The lower-level requirement(s) may closely resemble the higher-level one, or may be very different if the system engineers recognize a capability that the lower-level element must have in order to meet the higher-level requirement. In the latter case, the lower-level requirements are often referred to as *derived*.

The level of detail increases as we move down in the hierarchy. That is, system-level requirements are general in nature, while requirements at low levels in the hierarchy are very specific. A key part of the systems engineering approach to system development is *decomposition* and *abstraction:* the system is partitioned (decomposed) into finer and finer elements, while the requirements start at a highly abstract (general) level and become more specific for the lower-level elements. Large software-intensive systems are

among the most logically complex of human artifacts, and decomposition and abstraction are essential to the successful management of this complexity.

When flowdown is done, errors may be found in the allocation, the hierarchy definition, and the system requirements; thus the flowdown process is also iterative and may cause parts of previous processes to be repeated. Figure 8 shows the results of the first level of the flowdown process: a complete set of requirements for each of the subsystems. System-level requirement SYS001 was allocated to subsystems A and B; subsystem requirements (in this example, SSA001, SSA002, and SSB001) are written in response to the allocation. Similarly SYS002 was allocated to A and C, and subsystem requirements SSA003, SSA004, SSA005, SSC001, and SSC002 are the flowdown of SYS002 to subsystem levels. The result is a complete set of requirements for each of the subsystems. After completion of this level of flowdown, allocation of the subsystem requirements is carried out to the next level, followed by flowdown to that level. Again the processes are iterative and changes may be needed in the higher-level definition, allocation, or flowdown.

| SYSTEM REQUIREMENT | SYSTEM A REQUIREMENT | SYSTEM B REQUIREMENT | SYSTEM C REQUIREMENT |
|---|---|---|---|
| SYS 001 | SSA 001<br>SSA 002 | SSB 001 | — |
| SYS 002 | SSA 003<br>SSA 004<br>SSA 005 | — | — |
| SYS 003 | — | SSB 002<br>SSB 003 | — |
| SYS 004 | SSA 006<br>SSA 007 | SSB 004<br>SSB 005<br>SSB 006 | SSC 003 |
| SYS 005 | — | — | SSC 004<br>SSC 005 |
| SYS 006 | — | SSB 007<br>SSB 008 | — |
| SYS 007 | SSA 008<br>SSA 009 | SSB 009 | — |

**Figure 8**: **Example of flowdown of system requirements**

The process of partitioning, allocation, and flowdown is then repeated to as low a level as needed for this particular system; for software elements this is often to the module level. Figure 9 emphasizes the iterative nature of this process at each level in the many levels of partitioning, allocation, and flowdown.

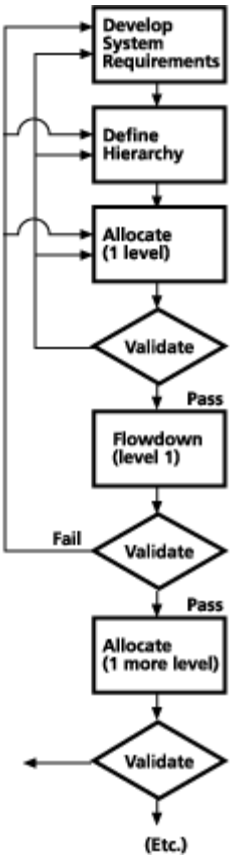


**Figure 9**: **Iteration in partitioning, allocation, and flowdown**

## Traceability

The number of requirements proliferates rapidly during the allocation and flowdown process. If we assume that there are four levels in the hierarchy, that each element partitions to four at the next level, that each requirement is allocated to two of the four, and that each flowdown results in three requirements per allocated element (all reasonable assumptions), there will be more than 250 requirements in the hierarchy for each system-level requirement. Keeping track of all these requirements is essential, to make sure that all requirements are properly flowed down to all levels, with no requirements lost and no "extras" thrown in. Reading and understanding the requirements to answer these questions is difficult enough; without a way to keep track of the flowdown path in a hierarchy of thousands of requirements, it becomes impossible. Traceability is the concept that implements the necessary bookkeeping.[19, 20] Establishment of traceability as allocation and flowdown are done helps ensure the validity of the process. Then, if changes are needed (such as a system-level requirement due to user input, or a lower-level requirement due to a problem with allocation,

flowdown, or feasibility), traceability enables the engineer to locate the related requirements, at higher and lower levels, that must be reviewed to see if they need to be changed.

Figure 10 shows the traceability path corresponding to the allocation and flowdown in Figures 7 and 8. System requirement SYS001 traces downward to SSA001, which in turn traces downward to PGA001, PGA002, and PGB001. SYS001 also traces to SSA002, which further traces to PGA003, PGC001, and PGC002. Upward traceability also exists, for example, PGB001 to SSA001 to SYS001. A similar hierarchy exists through SYS001's allocation and flowdown to subsystem B. Other formats such as trees and indented tables can be used to illustrate traceability, as in Dorfman and Flynn.[19]

Note that, while allocation and flowdown are technical tasks, traceability is not strictly an engineering function: it is a part of requirements management and is really only "bookkeeping." A case can be made that more of the requirements problems observed in system development are due to failures in requirements management than to technical functions. In the Software Engineering Institute's Capability Maturity Model® for Software,[22] these nontechnical aspects of requirements management are important enough that they are one of the six key process areas that a software development organization must satisfy to move beyond the first ad hoc or chaotic level of maturity.

| SYSTEM REQUIREMENT | SUBSYSTEM A REQUIREMENT | PROGRAM a REQUIREMENT | PROGRAM B REQUIREMENT | PROGRAM C REQUIREMENT |
|---|---|---|---|---|
| SYS 001 | SSA 001 | PGA 001<br>PGA 002 | PGB 001 | — |
| | SSA 002 | PGA 003 | — | PGC 001<br>PGC 002 |
| SYS 002 | SSA 003 | — | PGB 002<br>PGB 003 | — |
| | SSA 004 | PGA 004 | PCB 004 | PGC 003 |
| | SSA 005 | PGA 005 | PCB 005<br>PCB 006 | PGC 004<br>PGC 005 |

**Figure 10**: **The requirements traceability path**

## Interfaces

An additional step is interface definition. Before development of system requirements can begin, the system's external interfaces (the interfaces between the system and the outside world) must be known. As each level of partitioning, allocation, and flowdown takes place, the interfaces of each element to the rest of the system must be specified. This definition has two parts. First, interfaces defined at higher levels are made more specific; that is, the external interfaces to the entire system are identified as to which subsystem(s) actually perform the interface. Second, internal interfaces at that level are defined, that is, the subsystem-to-subsystem interfaces needed to enable each subsystem to meet the requirements allocated to it.

Figure 11 illustrates this concept. In the top diagram, A represents an external interface of the system, for example, an output produced by the system. When subsystems 1, 2, 3, and 4 are defined, as shown in the lower diagram, A is identified to subsystem 1, that is, the output originates in subsystem A, and internal interfaces, such as B between 3 and 4, are found to be necessary. This process continues throughout development of the hierarchy.

It is also possible that errors in partitioning, allocation, and flowdown will be discovered when interface definition is taking place, leading to iteration in those earlier steps.

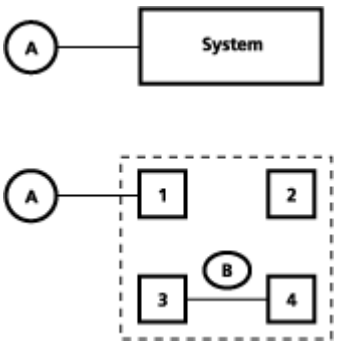


**Figure 11**: **Increasing detail of interface definition**

## The refined life cycle model

Next, let's examine the implications of the above processes for the life cycle models discussed earlier. It should now be apparent that the single "requirements analysis" phase, even if extended to "system requirements analysis" and "software requirements analysis," is inadequate. The life cycle model should account for the multiplicity of levels in the hierarchy, and furthermore should recognize that the various subsystems and other

elements at any level do not need to be synchronized. That is, if we are willing to accept the risk that flowdown to the last subsystem will surface some errors in partitioning or allocation, we can phase the subsystems, and of course the lower-level elements as well.

Figure 12 is the more realistic, and more complicated, life cycle chart that results from the above considerations. Although it builds on a baseline management approach, the nesting and phasing shown will apply to any of the models discussed earlier, or to combinations. The key features of Figure 12 are as follows:

1. For all but the lowest level of the hierarchy, the implementation phase of its life cycle becomes the entire development cycle for the next lower elements. Thus the subsystems have their development cycle shown as a phase of the system life cycle.
2. Elements at any level may be phased. Thus the subsystems are shown as starting and finishing their development cycles at different times. This approach has at least two advantages:

    - Staff can be employed more efficiently, since the schedules for each element can be phased to avoid excessive demand for any technical specialty at any time.
    - Integration can be carried out in a logical fashion, by adding one element at a time, rather than by the "big bang" approach of integrating all or many elements at the same time.

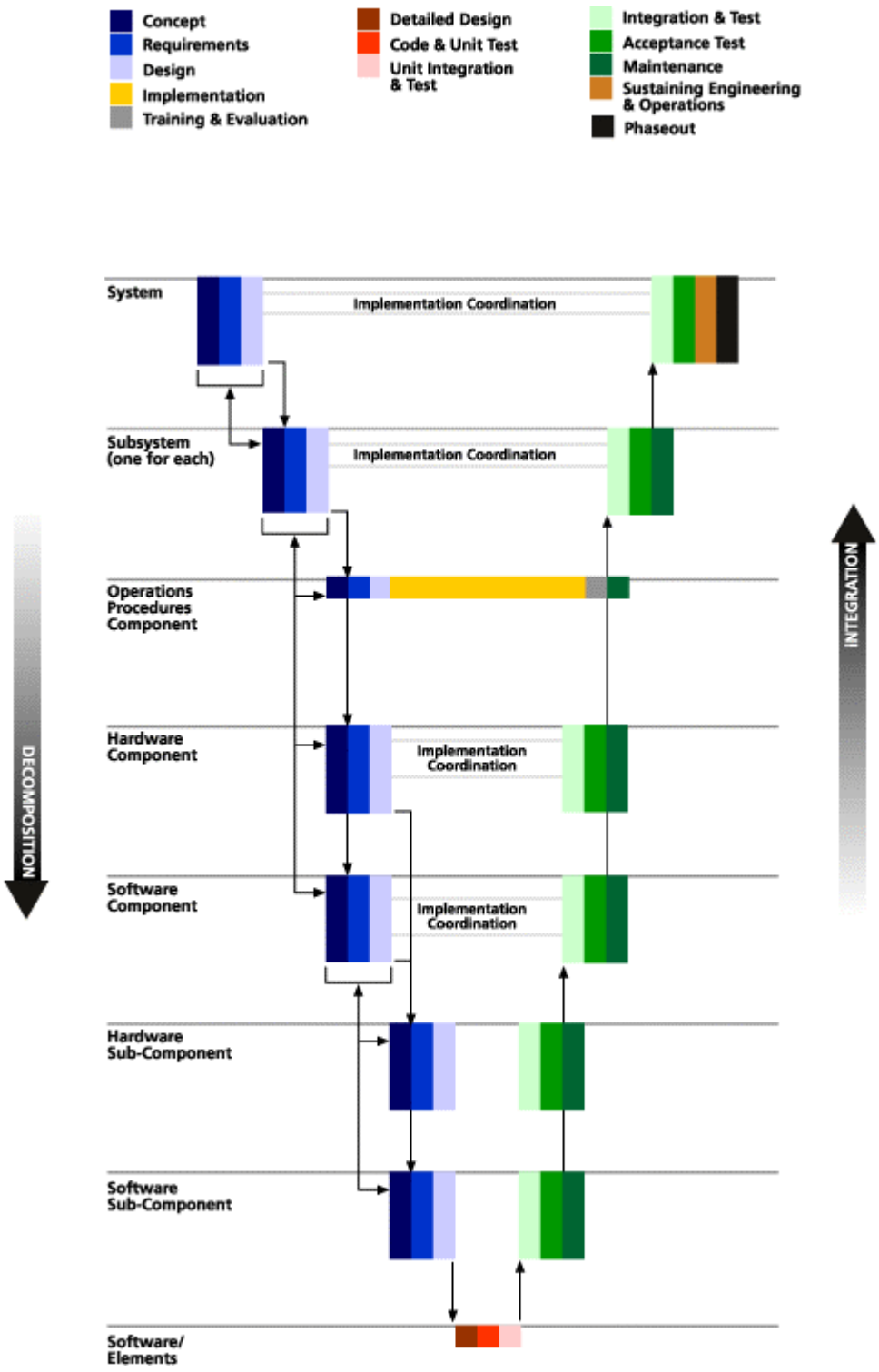


**Figure 12**: **Multilevel life cycle chart**

## Validation and verification

A final point in the framework of fundamentals relates to another requirements management task, the review of the requirements.[23] Validation and verification of the partitioning, allocation, flowdown, and interfaces are equally as important as their generation. It has been shown repeatedly that requirements errors not found until later in the development cycle are many times more expensive to fix than if they were found before the requirements phase was completed.[24] The baseline management model requires that all the requirements at all levels be fully and properly reviewed before design and implementation begin. The life cycle models wherein the requirements are not all determined and "frozen" before design begins specify review of requirements as they are considered ready to be the basis for some design and further development.

The attributes of a good requirements specification are addressed by Boehm.[23] Among the most important attributes are the following:

- Clear/unambiguous
- Complete
- Correct
- Understandable
- Consistent (internally and externally)
- Concise
- Feasible

It should be apparent that the evaluation of a requirements specification with respect to these attributes may be highly subjective and qualitative. Davis et al.[25] demonstrate the extreme difficulty of attempting to quantify the degree to which a specification exhibits these qualities. Nevertheless, these attributes are so important that the verification and validation of requirements against these criteria must be carried out, to the degree possible and as quantitatively as possible. Boehm [23] and Davis et al. [25] address approaches to validation and verification of requirements.

## Requirements engineering and architectural design

In the previous section, an overview is given of the process of partitioning, allocation, and flowdown. The result of this process (a definition of the hierarchy down to some level, generation of the requirements for all elements, and determination of the interfaces between them) is known as the *architectural design* or *top-level design* of the system. Although it is called a design, requirements engineering is involved throughout the process. What, then, is the distinction between requirements analysis and design in this process?

Requirements analysis is often defined as the "what" of a problem: implementation free; containing objectives, not methods. Design, then, is the "how": the implementation that will meet the requirements. The two are supposed to be kept distinct, although of course the feasibility of meeting a requirement always needs to be considered. However, if you look closely at the process of generating the architectural design, you will see that both requirements analysis and design are involved.[26]

The generation of system-level requirements is, to the extent possible, a pure "what," addressing the desired characteristics of the complete system. The next steps, determining the next level of the hierarchy and allocating system requirements to the elements, are in fact a "how": they do not address objectives beyond the system requirements, but they define a subsystem structure that enables the requirements to be met. Flowdown is again a "what," determining what each element should do (functions, performance, and so on).

Development of the architectural design is, then, a process in which the steps of requirements analysis and design alternate, with more detail being brought out at each cycle. The output of requirements analysis is input to the next stage of design, and the output of design is input to the next stage of requirements analysis.[27] If different people perform the two functions, one person's requirement is the next person's design, and one person's design is the next person's requirements.[28]

Pure requirements analysis, like pure design, can only go so far. Both disciplines are needed to achieve the desired result, a system that meets its user's needs. Note that the character of requirements analysis, like that of design, changes as we move down in the hierarchy: requirements analysis for a low-level element is much more detailed, and involves knowledge of previous design decisions. The tools and methods used for "analysis" do in fact support all aspects of architectural design development: partitioning, allocation, and flowdown. They therefore are useful in both the requirements analysis and design stages of the process.

## Requirements engineering practices

The principles of requirements engineering described above are valid and important, but for practical application additional specifics are needed. These specifics are provided by methods and tools. A *method*, sometimes referred to as a *methodology*, describes a general approach; a *tool*, usually but not always automated, provides a detailed, step-by-step approach to carrying out a method.

### Methods

Requirements analysis methods may be roughly divided into four categories, as shown in Figure 13. The categorizations should not be regarded as absolute: most methods have some of the characteristics of all the categories, but usually one viewpoint is primary.

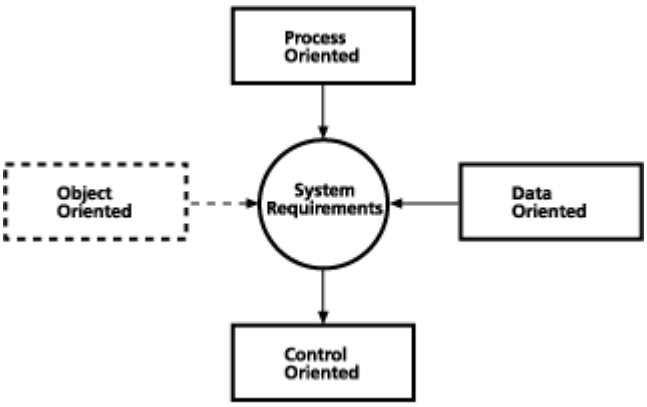


**Figure 13: Categories of requirements analysis methods**

Process-oriented methods take the primary viewpoint of the way the system transforms inputs into outputs, with less emphasis on the data itself and control aspects. Classical structured analysis (SA)[29] fits into this category, as do structured analysis and design technique (SADT),[30] and formal methods such as VDM[31] and Z.[32]

Data-oriented methods emphasize the system state as a data structure. While SA and SADT have secondary aspects of the data viewpoint, entity-relationship modeling[33] and JSD [7] are primarily data oriented.

Control-oriented methods emphasize synchronization, deadlock, exclusion, concurrence, and process activation and deactivation. SADT and the real-time extensions to SA[34, 35] are secondarily control oriented. Flowcharting is primarily process oriented.

Finally, object-oriented methods base requirements analysis on classes of objects of the system and their interactions with each other. Bailin[36] surveys the fundamentals of object-oriented analysis and describes the variations among the several different methods in use.

## Tools

The number of tools that support requirements engineering is growing rapidly, and even the most cursory survey is beyond the scope of this paper. Nevertheless, some discussion of the characteristics of requirements engineering tools, and trends in these characteristics, is in order.

Davis [18] has classified requirements tools as follows:

- Graphical editing
- Traceability
- Behavior modeling
- Databases and word processing—not designed for requirements engineering, but used in requirements applications
- Hybrid (combinations of the above)

Early requirements tools, such as SREM[37] and PSL/PSA,[38] were stand-alone, that is, were not integrated with any other tools, and supported the requirements analysis function by providing automation, either for a specific method or a group of methods. These tools consisted of hundreds of thousands of lines of code and ran on large mainframe computers. PSL/PSA used only very limited graphics in its early versions but later versions had improved graphics capabilities. SREM made heavy and effective use of graphics from the beginning. Both included some form of behavior modeling. In this same time frame, stand-alone traceability tools began to be developed. [19]

By the mid-1980s, integrated software development environments began to become available that, while they were more complex software products than the earlier tools, ran on powerful (usually UNIX-based) workstations that were smaller in size and lower in

cost than mainframes. Examples include Software through Pictures,[39] CADRE Teamwork, and similar products. These environments included requirements analysis tools, which generally supported one of the standard graphical analysis methods, and might also include traceability tools that enabled the requirements to be traced through design, implementation, and test. Some of the integrated environments provided a choice of analysis tools, such as tools that supported different methods. As computers continued to become smaller, more powerful, and lower in cost, full software development environments (known as computer-aided software engineering [CASE] or software engineering environments [SEE]) became available on desktop computers, and economics permitted providing such a computer to each member of the development team and networking them together.

CASE environments seemed full of promise during the late 1980s but somehow that promise has not been translated into pervasive use in the embedded systems and scientific software markets[40]; perhaps there was a perception that the environments supported a waterfall or baseline management model of end-to-end software development better than the prototyping, evolutionary, or incremental models that became increasingly popular. The requirements tools that are part of these integrated environments are, of course, limited to the market penetration of the environments themselves. In the application area of transaction-oriented systems such as financial and database, tools implementing an entity-relationship model such as Texas Instruments's Information Engineering Facility have revolutionized the way software is developed, but in the scientific and embedded application areas stand-alone tools, if any, are used. Perhaps the current trend toward object-oriented technologies will revitalize the market for tools and for integrated tool environments.

## The role of tools and methods

In conclusion, a few words are in order about the role of tools and methods in requirements engineering, and indeed across the full scope of software and systems engineering. Proper use of tools and methods is an important part of process maturity as advocated by the Software Engineering Institute, [22] and process maturity has been shown to lead to improvements in a software development organization's productivity and quality.[41] It has equally been shown that developers cannot adopt or purchase methods and tools and just throw them at the problem. Selection of tools and methods requires study of their applicability and compatibility with the organization's practices. A process must be identified that meets the organization's needs; methods must be selected that support the process, with tool availability one of the factors to be considered; then and only then should tools be purchased. Commitment, training, and the time to make the transition are essential. Tools and methods are not a panacea: if selected and used

correctly they can provide great benefits; if regarded as magic solutions or otherwise misused, they will prove an expensive failure.

## References


[1] Naur, P., and B. Randell, eds., Software Engineering: Report on a Conference Sponsored by the NATO Science Commission, Garmisch, Germany, 7–11 October 1968. Scientific Affairs Division, NATO, Brussels, January 1969.

[2] Alford, M. W., and J. T. Lawson, "Software Requirements Engineering Methodology (Development)." RADC-TR-79-168, U.S. Air Force Rome Air Development Center, Griffiss AFB, NY, June 1979 (DDC-AD-A073132).

[3] Schwartz, J.I., "Construction of Software, Problems and Practicalities," in *Practical Strategies for Developing Large Software Systems*, E. Horowitz, ed., Addison-Wesley, Reading, MA, 1975.

[4] IEEE Standard 610.12-1990, IEEE Standard Glossary of Software Engineering Terminology, IEEE, NY, 1990.

[5] Boehm, Barry W., "A Spiral Model of Software Development and Enhancement," *Computer*, Vol. 21, No. 5, May 1988, pp. 61-72.

[6] Davis, Alan M., Edward H. Bersoff, and Edward R. Comer, "A Strategy for Comparing Alternative Software Development Life Cycle Models," *IEEE Transactions on Software Engineering*, Vol. 14, No. 10, October 1988, pp. 1453-1461.

[7] Cameron, John R., "An Overview of JSD," *IEEE Transactions on Software Engineering*, Vol. 12, No. 2, February 1986, pp. 222-240.

[8] Royce, Winston W., "Managing the Development of Large Software Systems," Proceedings, IEEE Wescon, August 1970. Reprinted in *Proceedings, 9th International Conference on Software* Engineering (Monterey, CA, March 30-April 2, 1987), IEEE Computer Society Press, Washington, DC, 1987, pp. 328-338.

[9] McCracken, Daniel D., and Michael A. Jackson, "Life Cycle Concept Considered Harmful," *ACM Software Engineering Notes*, Vol. SE-7, No. 2, 1982, pp. 29–32.

[10] Gladden, G. R., "Stop the Life Cycle, I Want to Get Off," *ACM Software Engineering Notes*, Vol. SE-7, No. 2, April 1982, pp. 35–39.

[11] Brooks, Frederick P., Jr., Chairman, *Report of the Defense Science Board Task Force on Military Software*, Office of the Under Secretary of Defense for Acquisition, U.S. Department of Defense, Washington, DC, September 1987.

[12] Gomaa, Hassan, and D.B.H. Scott, "Prototyping as a Tool in the Specification of User Requirements," *Proceedings, Fifth International Conference on Software Engineering (San Diego, CA, March 9-12, 1981)*, IEEE Computer Society Press, Washington, DC, 1981, pp. 333-342.

[13] Verner, June, and Graham Tate, "Third-Generation versus Fourth-Generation Software Development," *IEEE Software*, Vol. 5, No. 4, July 1988, pp. 8-14.

[14] Cobb, R. H., "In Praise of 4GLs," *Datamation*, July 15, 1985, pp. 36–46.

[15] Boehm, Barry W., ed., *Software Risk Management*, IEEE Computer Society Press, Los Alamitos, CA, 1989, p. 434.

[16] Fairley, Richard E., and Richard H. Thayer, "The Concept of Operations: The Bridge from Operational Requirements to Technical Specifications," in *Software Engineering*, M. Dorfman and R.H. Thayer, eds., IEEE Computer Society Press, Los Alamitos, CA, 1997.

[17] *Acquisition Software Development Capability Evaluation* (2 volumes), AFMC Pamphlet 63-103, Department of the Air Force, HQ Air Force Materiel Command, Wright-Patterson AFB, OH, June 15, 1994.

[18] Davis, Alan M., private communication, 1996.

[19] Dorfman, Merlin, and Richard F. Flynn, "ARTS—an Automated Requirements Traceability System," *Journal of Systems and Software*, Vol. 4, No. 1, 1984, pp. 63-74.

[20] Palmer, James D., "Traceability," in *Software Engineering*, M. Dorfman and R.H. Thayer, eds., IEEE Computer Society Press, Los Alamitos, CA, 1997, pp. 266-276.

[21] Goguen, Joseph A., and Charlotte Linde, "Techniques for Requirements Elicitation," *Proceedings of the International Symposium on Requirements Engineering (Colorado Springs, CO, April 18-22)* IEEE Computer Society Press, Los Alamitos, CA, 1993.

[22] Paulk, Mark C., et al., *Capability Maturity Model® for Software, Version 1.1*, CMU/SEI-93-TR-24, Software Engineering Institute, Carnegie Mellon University, Pittsburgh, PA, February 1993. See also Paulk, Mark C., et al., *Key Practices of the Capability Maturity Model® for Software, Version 1.1*, CMU/SEI-93-TR-25, Carnegie Mellon University, Pittsburgh, PA, February 1993. Available at http://www.sei.cmu.edu/publications/documents/93.reports/93.tr.024.html and http://www.sei.cmu.edu/publications/documents/93.reports/93.tr.025.html

[23] Boehm, Barry W., "Verifying and Validating Software Requirements and Design Specifications," *IEEE Software*, Vol. 1, No. 1, January 1984, pp. 75-88.

[24] Boehm, Barry W., *Software Engineering Economics*, Prentice-Hall, Englewood Cliffs, NJ, 1981.

[25] Davis, Alan, et al., "Identifying and Measuring Quality in a Software Requirements Specification," *Proceedings of the First International Software Metrics Symposium* (Baltimore, MD, May 21-22) IEEE Computer Society Press, Los Alamitos, CA, 1993.

[26] Swartout, W., and R. Balzer, "On the Inevitable Intertwining of Specification and Design," *Communications of the ACM*, Vol. 27, No. 7, July 1982, pp. 438-440.

[27] Hatley, Derek J., and Imtiaz A. Pirbhai, *Strategies for Real-Time System Specification*, Dorset House, NY, 1987.

[28] Davis, Alan M., *Software Requirements: Objects, Functions, and States*, Prentice-Hall, Englewood Cliffs, NJ, 1993.

[29] Svoboda, Cyril P., "Tutorial on Structured Analysis," in *System and Software Requirements Engineering*, R.H. Thayer and M. Dorfman, eds., IEEE Computer Society Press, Los Alamitos, CA, 1990.

[30] Ross, Douglas T., "Structured Analysis (SA): A Language for Communicating Ideas," *IEEE Transactions on Software Engineering*, Vol. 3, No. 1, January 1977, 16-33.

[31] Bjoerner, Dines, "On the Use of Formal Methods in Software Development," *Proceedings, 9th International Conference on Software* Engineering (Monterey, CA, March 30-April 2, 1987), IEEE Computer Society Press, Washington, DC, 1987, pp. 17-29.

[32] Norris, M., "Z (A Formal Specification Method). A Debrief Report," STARTS, National Computing Centre, Ltd., 1986. Reprinted in *System and Software Requirements Engineering*, R.H. Thayer and M. Dorfman, eds., IEEE Computer Society Press, Los Alamitos, CA, 1990.

[33] Reilly, John R., "Entity-Relationship Approach to Data Modeling," in *Software Requirements Engineering*, 2nd ed., R.H. Thayer and M. Dorfman, eds., IEEE Computer Society Press, Los Alamitos, CA, 1997.

[34] Ward, Paul T., and Stephen J. Mellor, *Structured Development Techniques for Real-Time Systems* (3 vols.). Prentice-Hall, Englewood Cliffs, NJ, 1985.

[35] Hatley, Derek J., "The Use of Structured Methods in the Development of Large Software-Based Avionics Systems," AIAA Paper 84-2592, Sixth Digital Avionics Systems Conference (Baltimore, MD, December 3-6), AIAA, NY, 1984.

[36] Bailin, C.A., "Object Oriented Requirements Analysis," in *Encyclopedia of Software Engineering*, John J. Marciniak, ed., John Wiley & Sons, NY, 1994.

[37] Alford, Mack W., "SREM at the Age of Eight: The Distributed Computing Design System," *Computer*, Vol. 18, No. 4, April 1985, 36-46.

[38] Sayani, Hassan, "PSL/PSA at the Age of Fifteen: Tools for Real-Time and Non-Real-Time Analysis," in *System and Software Requirements Engineering*, R.H. Thayer and M. Dorfman, eds., IEEE Computer Society Press, Los Alamitos, CA, 1990, 403-417.

[39] Wasserman, Anthony I., and P. A. Pircher, "A Graphic, Extensible Integrated Environment for Software Development," *ACM SIGPLAN Notices*, Vol. 12, No. 1, January 1987, pp. 131-142.

[40] Lewis, Ted, "The Big Software Chill," *Computer*, Vol. 29, No. 3, March 1996, pp. 12-14.

[41] Herbsleb, James, et al., *Benefits of CMM-Based Software Process Improvement: Initial Results*, CMU/SEI-94-TR-13, Software Engineering Institute, Carnegie Mellon University, Pittsburgh, PA, August 1994. Available at http://www.sei.cmu.edu/publications/documents/94.reports/94.tr.013.html

## About the author

Merlin Dorfman is a technical consultant in the Space Systems Product Center, Lockheed Martin Missiles and Space Company, Sunnyvale, Calif. He specializes in systems

engineering for software-intensive systems, software process improvement, and algorithm development for data processing systems. Merlin Dorfman has no affiliation with the Software Engineering Institute.

The views expressed in this article are the author's only and do not represent directly or imply any official position or view of the Software Engineering Institute or Carnegie Mellon University. This article is intended to stimulate further discussion about this topic.

The Software Engineering Institute (SEI) is a federally funded research and development center sponsored by the U.S. Department of Defense and operated by Carnegie Mellon University.