A Spiral Model of Software Development and Enhancement

Barry W. Boehm, TRW Defense Systems Group

"Stop the life cycle—I want to get off!"
"Life-cycle Concept Considered Harmful."
" The waterfall model is dead."
"No, it isn't, but it should be."

These statements exemplify the current debate about software life-cycle process models. The topic has recently received a great deal of attention.

The Defense Science Board Task Force Report on Military Software[1] issued in 1987 highlighted the concern that traditional software process models were discouraging more effective approaches to software development such as prototyping and software reuse. The Computer Society has sponsored tutorials and workshops on software process models that have helped clarify many of the issues and stimulated advances in the field (see "Further Reading").

The spiral model presented in this article is one candidate for improving the software process model situation. The major distinguishing feature of the spiral model is that it creates a *risk-driven* approach to the software process rather than a primarily *document-driven* or *code-driven* process. It incorporates many of the strengths of other models and resolves many of their difficulties.

This article opens with a short description of software process models and the issues they address. Subsequent sections outline the process steps involved in the spiral model; illustrate the application of the spiral model to a software project, using the TRW Software Productivity Project as an example; summarize the primary advantages and implications involved in using the spiral model and the primary difficulties in using it at its current incomplete level of elaboration; and present resulting conclusions.

Background on software process models

The primary functions of a software process model are to determine the *order of the stages* involved in software development and evolution and to establish the *transition criteria* for progressing from one stage to the next. These include completion criteria for the current stage plus choice criteria and entrance criteria for the next stage. Thus, a process model addresses the following software project questions:

(1) What shall we do next?

(2) How long shall we continue to do it'?

Consequently, a process model differs from a software method (often called a methodology) in that a method's primary focus is on how to navigate through each phase (determining data, control, or "uses" hierarchies; partitioning functions; allocating requirements) and how to represent phase products (structure charts; stimulus—response threads; state transition diagrams).

Why are software process models important? Primarily because they provide guidance on the order (phases, increments, prototypes, validation tasks, etc.) in which a project should carry out its major tasks. Many software projects, as the next section shows, have come to grief because they pursued their various development and evolution phases in the wrong order.

Evolution of process models. Before concentrating in depth on the spiral model, we should take a look at a number of others: the code-and-fix model, the stage-wise model, the waterfall model, the evolutionary development model, and the transform model.

The code-and-fix model. The basic model used in the earliest days of software development contained two steps:

(1) Write some code.

(2) Fix the problems in the code.

Thus, the order of the steps was to do some coding first and to think about the requirements, design, test, and maintenance later. This model has three primary difficulties:

(a) After a number of fixes, the code became so poorly structured that subsequent fixes were very expensive. This underscored the need for a design phase prior to coding.

(b) Frequently, even well-designed software was such a poor match to users' needs that it was either rejected outright or expensively redeveloped. This made the need for a requirements phase prior to design evident.

(c) Code was expensive to fix because of poor preparation for testing and modification. This made it clear that explicit recognition of these phases, as well as test and evolution planning and preparation tasks in the early phases, were needed.

The stagewise and waterfall models. As early as 1956, experience on large software systems such as the Semi-Automated Ground Environment (SAGE) had led to the recognition of these problems and to the development of a stagewise model[2] to address them. This model stipulated that software be developed in successive stages (operational plan, operational specifications, coding specifications, coding, parameter testing, assembly testing, shakedown, and system evaluation).

The waterfall model,[3] illustrated in Figure 1, was a highly influential 1970 refinement of the stagewise model. It provided two primary enhancements to the stagewise model:

(1) Recognition of the feedback loops between stages, and a guideline to confine the feedback loops to successive stages to minimize the expensive rework involved in feedback across many stages.

(2) An initial incorporation of prototyping in the software life cycle, via a "build it twice" step running in parallel with requirements analysis and design.

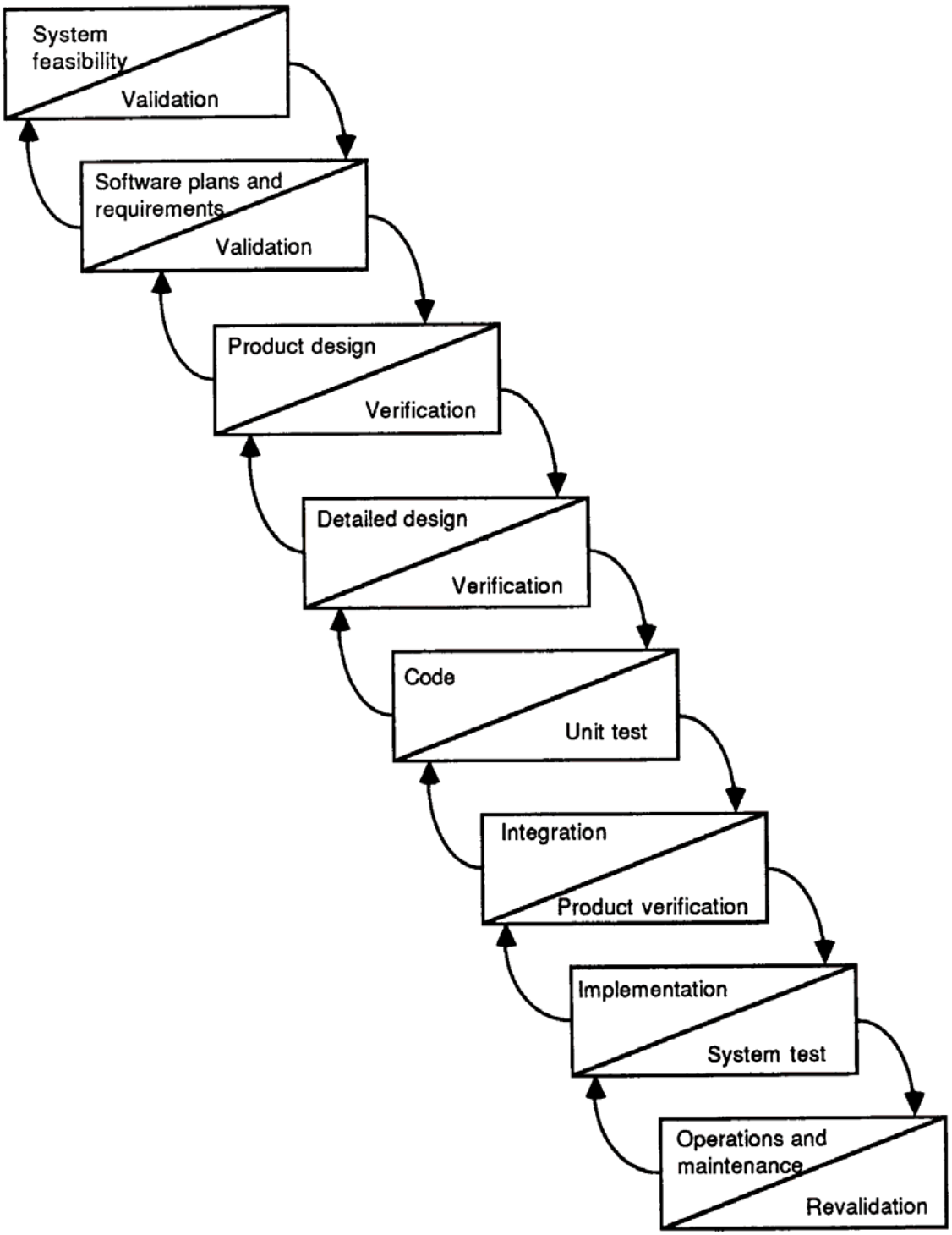

Figure 1. The waterfall model of the software life cycle.

The waterfall model's approach helped eliminate many difficulties previously encountered on software projects. The waterfall model has become the basis for most software acquisition standards in government and industry. Some of its initial difficulties have been addressed by adding extensions to cover incremental development, parallel

developments, program families, accommodation of evolutionary changes, formal software development and verification, and stagewise validation and risk analysis.

However, even with extensive revisions and refinements, the waterfall model's basic scheme has encountered some more fundamental difficulties, and these have led to the formulation of alternative process models.

A primary source of difficulty with the waterfall model has been its emphasis on fully elaborated documents as completion criteria for early requirements and design phases. For some classes of software, such as compilers or secure operating systems, this is the most effective way to proceed. However, it does not work well for many classes of software, particularly interactive end-user applications. Document-driven standards have pushed many projects to write elaborate specifications of poorly understood user interfaces and decision support functions, followed by the design and development of large quantities of unusable code.

These projects are examples of how waterfall-model projects have come to grief by pursuing stages in the wrong order. Furthermore, in areas supported by fourth-generation languages (spreadsheet or small business applications), it is clearly unnecessary to write elaborate specifications for one's application before implementing it.

The evolutionary development model. The above concerns led to the formulation of the *evolutionary development* model,[4] whose stages consist of expanding increments of an operational software product, with the directions of evolution being determined by operational experience.

The evolutionary development model is ideally matched to a fourth-generation language application and well matched to situations in which users say, "I can't tell you what I want, but I'll know it when I see it." It gives users a rapid initial operational capability and provides a realistic operational basis for determining subsequent product improvements.

Nonetheless, evolutionary development also has its difficulties. It is generally difficult to distinguish it from the old code-and-fix model, whose spaghetti code and lack of plan-fling were the initial motivation for the waterfall model. It is also based on the often-unrealistic assumption that the user's operational system will be flexible enough to accommodate unplanned evolution paths. This assumption is unjustified in three primary circumstances:

(1) Circumstances in which several independently evolved applications must subsequently be closely integrated.

(2) "Information-sclerosis" cases, in which temporary workarounds for software deficiencies increasingly solidify into unchangeable constraints on evolution. The following comment is a typical example: "It's nice that you could change those equip-ment codes to make them more intelligible for us, but the Codes Committee just met and established the current codes as company standards."

(3) Bridging situations, in which the new software is incrementally replacing a large existing system. If the existing system is poorly modularized, it is difficult to provide a good sequence of "bridges" between the old software and the expanding increments of new software.

Under such conditions, evolutionary development projects have come to grief by pursuing stages in the wrong order: evolving a lot of hard-to-change code before addressing long-range architectural and usage considerations.

The transform model. The "spaghetti code" difficulties of the evolutionary development and code-and-fix models can also become a difficulty in various classes of waterfall-model applications, in which code is optimized for performance and becomes increasingly hard to modify. The transform model[5] has been proposed as a solution to this dilemma.

The transform model assumes the existence of a capability to automatically convert a formal specification of a software product into a program satisfying the specification. The steps then prescribed by the transform model are

- a formal specification of the best initial understanding of the desired product;
- automatic transformation of the specification into code;
- an iterative loop, if necessary, to improve the performance of the resulting code by giving optimization guidance to the transformation system;
- exercise of the resulting product; and
- an outer iterative loop to adjust the specification based on the resulting operational experience, and to rederive, reoptimize, and exercise the adjusted software product.

The transform model thus bypasses the difficulty of having to modify code that has be-come poorly structured through repeated reoptimizations, since the modifications are made to the specification. It also avoids the extra time and expense involved in the inter-mediate design, code, and test activities.

Still, the transform model has various difficulties. Automatic transformation capabilities are only available for small products in a few limited areas: spreadsheets, small fourth-generation language applications, and limited computer science domains. The transform model also shares some of the difficulties of the evolutionary development model, such as the assumption that users' operational systems will always be flexible enough to support unplanned evolution paths. Additionally, it would face a formidable knowledge-base-maintenance problem in dealing with the rapidly increasing and evolving supply of reusable software components and commercial software products. (Simply consider the problem of tracking the costs, performance, and features of all

commercial database management systems, and automatically choosing the best one to implement each new or changed specification.)

The spiral model

The spiral model of the software process (see Figure 2) has been evolving for several years, based on experience with various refinements of the waterfall model as applied to large government software projects. As will be discussed, the spiral model can accommodate most previous models as special cases and further provides guidance as to which combination of previous models best fits a given software situation. Development of the TRW Software Productivity System (TRW-SPS), described in the next section, is its most complete application to date.

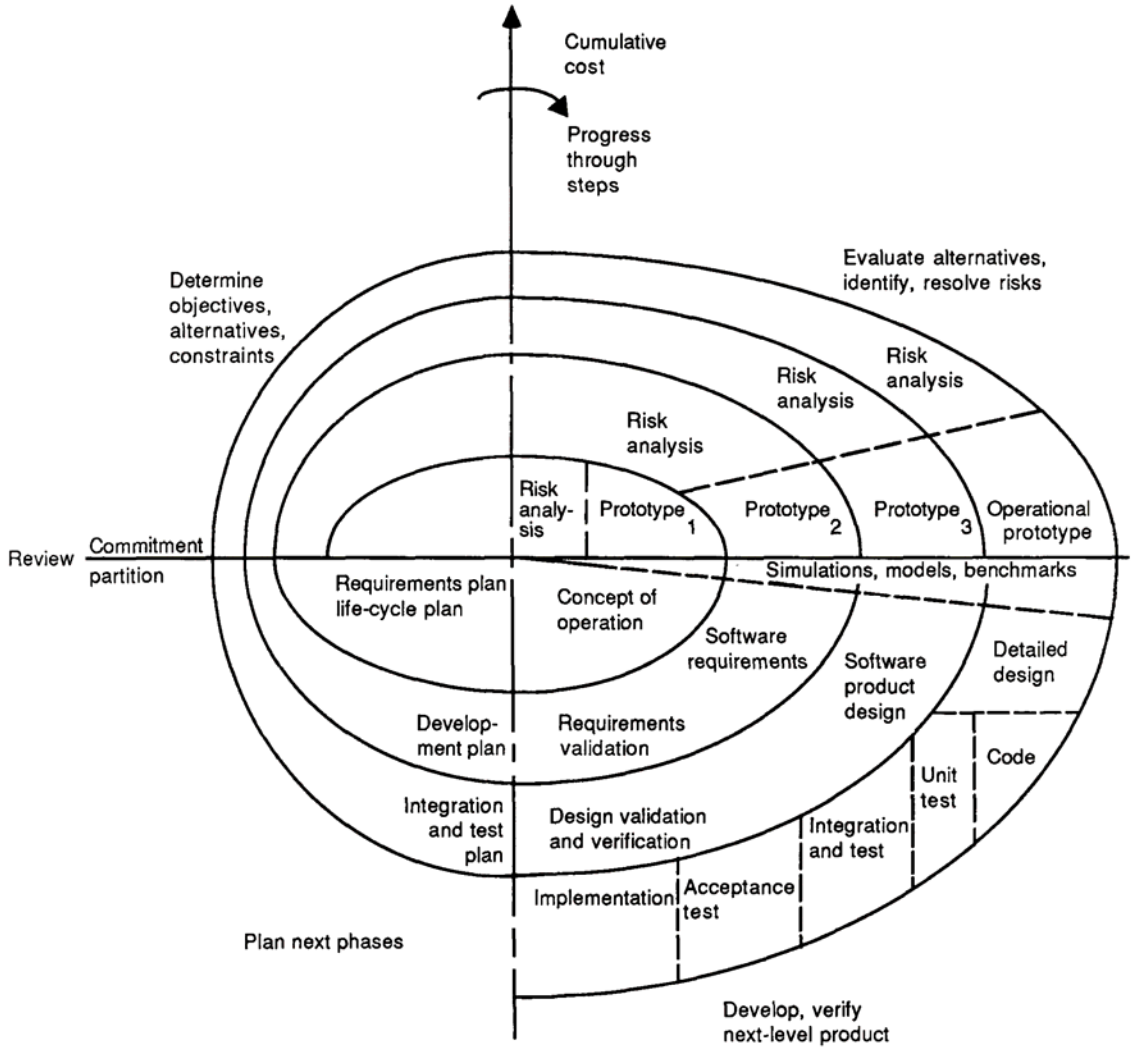

Figure 2. Spiral model of the software process.

The radial dimension in Figure 2 represents the cumulative cost incurred in accomplishing the steps to date; the angular dimension represents the progress made in

completing each cycle of the spiral. (The model reflects the underlying concept that each cycle involves a progression that addresses the same sequence of steps, for each portion of the product and for each of its levels of elaboration, from an overall concept of operation document down to the coding of each individual program.) Note that some artistic license has been taken with the increasing cumulative cost dimension to enhance legibility of the steps in Figure 2.

A typical cycle of the spiral. Each cycle of the spiral begins with the identification of

- the objectives of the portion of the product being elaborated (performance, functionality, ability to accommodate change, etc.);
- the alternative means of implementing this portion of the product (design A , design B, reuse, buy, etc.); and
- the constraints imposed on the application of the alternatives (cost, schedule, inter-face, etc.).

The next step is to evaluate the alternatives relative to the objectives and constraints. Frequently, this process will identify areas of uncertainty that are significant sources of project risk. If so, the next step should involve the formulation of a cost-effective strategy for resolving the sources of risk. This may involve prototyping, simulation, benchmarking, reference checking, administering user questionnaires, analytic modeling, or combinations of these and other risk resolution techniques.

Once the risks are evaluated, the next step is determined by the relative remaining risks. If performance or user-interface risks strongly dominate program development or internal interface-control risks, the next step may be an evolutionary development one: a minimal effort to specify the overall nature of the product, a plan for the next level of prototyping, and the development of a more detailed prototype to continue to resolve the ma-jor risk issues.

If this prototype is operationally useful and robust enough to serve as a low-risk base for future product evolution, the subsequent risk-driven steps would be the evolving series of evolutionary prototypes going toward the right in Figure 2. In this case, the option of writing specifications would be addressed but not exercised. Thus, risk considerations can lead to a project implementing only a subset of all the potential steps in the model.

On the other hand, if previous prototyping efforts have already resolved all of the performance or user-interface risks, and program development or interface-control risks dominate, the next step follows the basic waterfall approach (concept of operation, soft-ware requirements, preliminary design, etc. in Figure 2), modified as appropriate to incorporate incremental development. Each level of software specification in the figure is then followed by a validation step and the preparation of plans for the succeeding cycle.

In this case, the options to prototype, simulate, model, and so on are addressed but not exercised, leading to the use of a different subset of steps.

This risk-driven subsetting of the spiral model steps allows the model to accommodate any appropriate mixture of a specification-oriented, prototype-oriented, simulation-oriented, automatic transformation-oriented, on other approach to software development. In such cases, the appropriate mixed strategy is chosen by considering the relative magnitude of the program risks and the relative effectiveness of the various techniques in resolving the risks. In a similar way, risk-management considerations can determine the amount of time and effort that should be devoted to such other project activities as plan-fling, configuration management, quality assurance, formal verification, and testing. In particular, risk-driven specifications (as discussed in the next section) can have varying degrees of completeness, formality, and granularity, depending on the relative risks of doing too little or too much specification.

An important feature of the spinal model, as with most other models, is that each cycle is completed by a review involving the primary people or organizations concerned with the product. This review covers all products developed during the previous cycle, including the plans for the next cycle and the resources required to carry them out. The review's major objective is to ensure that all concerned parties are mutually committed to the approach for the next phase.

The plans for succeeding phases may also include a partition of the product into increments for successive development or components to be developed by individual organizations or persons. For the latter case, visualize a series of parallel spinal cycles, one for each component, adding a third dimension to the concept presented in Figure 2. For example, separate spirals can be evolving for separate software components or increments. Thus, the review-and-commitment step may range from an individual walk-through of the design of a single programmer's component to a major requirements review involving developer, customer, user, and maintenance organizations.

Initiating and terminating the spiral. Four fundamental questions arise in considering this presentation of the spiral model:

(1) How does the spiral ever get started?

(2) How do you get off the spiral when it is appropriate to terminate a project early?

(3) Why does the spiral end so abruptly?

(4) What happens to software enhancement (or maintenance)?

The answers to these questions involve an observation that the spinal model applies equally well to development or enhancement efforts. In either case, the spiral gets started by a hypothesis that a particular operational mission (or set of missions) could be improved by a software effort. The spiral process then involves a test of this hypothesis: at any time, if the hypothesis fails the test (for example, if delays cause a software

product to miss its market window, or if a superior commercial product becomes available), the spirat is terminated. Otherwise, it terminates with the installation of new on modified soft-ware, and the hypothesis is tested by observing the effect on the operational mission. Usually, experience with the operational mission leads to further hypotheses about software improvements, and a new maintenance spiral is initiated to test the hypothesis. Initiation, termination, and iteration of the tasks and products of previous cycles are thus implicitly defined in the spiral model (although they're not included in Figure 2 to simplify its presentation).

Using the spiral model

The various rounds and activities involved in the spinal model are best understood through use of an example. The spiral model was used in the definition and development of the TRW Software Productivity System (TRW-SPS), an integrated software engineering environment.[6] The initial mission opportunity coincided with a corporate initiative to improve productivity in all appropriate corporate operations and an initial hypothesis that software engineering was an attractive area to investigate. This led to a small, extra "Round 0" circuit of the spiral to determine the feasibility of increasing software productivity at a reasonable corporate cost. (Very large or complex software projects will frequently precede the "concept of operation" round of the spiral with one on more smaller rounds to establish feasibility and to reduce the range of alternative solutions quickly and inexpensively.)

Tables 1, 2, and 3 summarize the application of the spiral model to the first three rounds of defining the SPS. The major features of each round are subsequently discussed and are followed by some examples from later rounds, such as preliminary and detailed design.

Table 1. Spiral model usage: TRW Software Productivity System, Round 0

Objectives	Significantly increase software productivity
Constraints	At reasonable cost Within context of TRW culture • Government contracts, high tech, people oriented, security
Alternatives	Management: Project organization, policies, planning, control Personnel: Staffing, incentives, training Technology: Tools, workstations, methods, reuse Facilities: Offices, communications
Risks	May be no high-leverage improvements Improvements may violate constraints
Risk resolution	Internal surveys Analyze cost model Analyze exceptional projects Literature search
Risk resolution results	Some alternatives infeasible • Single time-sharing system: Security Mix of alternatives can produce significant gains

	• Factor of two in five years Need further study to determine best mix
Plan for next phase	Six-person task force for six months More extensive surveys and analysis • Internal, external, economic Develop concept of operation, economic rationale
Commitment	Fund next phase

Round 0: Feasibility study. This study involved five part-time participants over a two-to-three-month period. As indicated in Table 1 , the objectives and constraints were expressed at a very high level and in qualitative terms like "significantly increase," "at reasonable cost," etc.

Some of the alternatives considered, primarily those in the "technology" area, could lead to development of a software product, but the possible attractiveness of a number of non-software alternatives in the management, personnel, and facilities areas could have led to a conclusion not to embank on a software development activity.

The primary risk areas involved possible situations in which the company would invest a good deal only to find that

- Resulting productivity gains were not significant
- Potentially high-leverage improvements were not compatible with some aspects of the "TRW culture"

The risk-resolution activities undertaken in Round 0 were primarily surveys and analyses, including structured interviews of software developers and managers; an initial analysis of productivity leverage factors identified by the constructive cost model (COCOMO);[7] and an analysis of previous projects at TRW exhibiting high levels of productivity.

The risk analysis results indicated that significant productivity gains could be achieved at a reasonable cost by pursuing an integrated set of initiatives in the four major areas. However, some candidate solutions, such as a software support environment based on a single, corporate, maxicomputer-based time-sharing system, were found to be in conflict with TRW constraints requiring support of different levels of security-classified projects. Thus, even at a very high level of generality of objectives and constraints, Round 0 was able to answer basic feasibility questions and eliminate significant classes of candidate solutions.

The plan for Round 1 involved commitment of 12 man-months compared to the two man-months invested in Round 0 (during these rounds, all participants were part-time). Round 1 here corresponded fairly well to the initial round of the spinal model shown in Figure 2, in that its intent was to produce a concept of operation and a basic life-cycle plan for implementing whatever preferred alternative emerged.

Table 2. Spiral model usage: TRW Software Productivity System, Round 1

Objectives	Double software productivity in five years
Constraints	$10,000 per person investment Within context of TRW culture • Government contracts, high tech, people oriented, security Preference for TRW products
Alternatives	Office: Private/modular/… Communication: LAN/star/concentrators/… Terminals: Private/shared; smart/dumb Tools: SREM/PSL-PSA/…; PDL/SADT/… CPU: IBM/DEC/CDC/…
Risks	May miss high-leverage options TRW LAN price/performance Workstation cost
Risk resolution	Extensive external surveys, visits TRW LAN benchmarking Workstation price projections
Risk resolution results	Operations concept: Private offices, TRW LAN, personal terminals, VAX Begin with primarily dumb terminals; experiment with smart workstations Defer operating system, tools selection
Plan for next phase	Partition effort into software development environment (SDE), facilities, management Develop first-cut, prototype SDE • Design-to-cost: 15-person team for one year Plan for external usage
Commitment	Develop prototype SDE Commit an upcoming project to use SDE Commit the SDE to support the project Form representative steering group

Round 1: Concept of operations. Table 2 summarizes Round 1 of the spinal along the lines given in Table 1 for Round 0. The features of Round 1 compare to those of Round 0 as follows:

- The level of investment was greater (12 versus 2 man-months).
- The objectives and constraints were more specific ("double software productivity in five years at a cost of $10,000 a person" versus "significantly increase productivity at a reasonable cost").
- Additional constraints surfaced, such as the preference for TRW products [particularly, a TRW-developed local area network (LAN) system].

- The alternatives were more detailed ("SREM, PSL/PSA on SADT, as requirements tools, etc." versus "tools"; "private/shared" terminals, "smart/dumb" terminals versus "workstations").

- The risk areas identified were more specific ("TRW LAN price-performance within a $10,000-per-person investment constraint" versus "improvements may violate reasonable-cost constraint").

- The risk-resolution activities were more extensive (including the benchmarking and analysis of a prototype TRW LAN being developed for another project).

- The result was a fairly specific operational concept document, involving private of-flees tailored to software work patterns and personal terminals connected to VAX superminis via the TRW LAN. Some choices were specifically deferred to the next round, such as the choice of operating system and specific tools.

- The life-cycle plan and the plan for the next phase involved a partitioning into separate activities to address management improvements, facilities development, and development of the first increment of a software development environment.

- The commitment step involved more than just an agreement with the plan. It committed to apply the environment to an upcoming 100-person testbed software project and to develop an environment focusing on the testbed project's needs. It also specified forming a representative steering group to ensure that the separate activities were well-coordinated and that the environment would not be overly optimized around the testbed project.

Although the plan recommended developing a prototype environment, it also recommended that the project employ requirements specifications and design specifications in a risk-driven way. Thus, the development of the environment followed the succeeding rounds of the spiral model.

Table 3. Spiral model usage: TRW Software Productivity System, Round 2.

Objectives	User-friendly system Integrated software, office-automation tools Support all project personnel Support all life-cycle phases
Constraints	Customer-deliverable SDE ⇨ Portability Stable, reliable service
Alternatives	OS: VMS/AT&T Unix/Berkeley Unix/ISC Host-target/fully portable tool set Workstations: Zenith/LSI-11/…
Risks	Mismatch to user-project needs, priorities

	User-unfriendly system • 12-language syndrome; experts-only Unix performance, support Workstation/mainframe compatibility
Risk resolution	User-project surveys, requirements participation Survey of Unix-using organizations Workstation study
Risk resolution results	Top-level requirements specification Host-target with Unix host Unix-based workstations Build user-friendly front end for Unix Initial focus on tools to support early phases
Plan for next phase	Overall development plan • for tools: SREM, RTT, PDL, office automation tools • for front end: Support tools • for LAN: Equipment, facilities
Commitment	Proceed with plans

Round 2: Top-Level Requirements Specification. Table 3 shows the corresponding steps involved during Round 2 defining the software productivity system. Round 2 decisions and their rationale were covered in earlier work[6]; here, we will summarize the considerations dealing with risk management and the use of the spiral model:

- The initial risk-identification activities during Round 2 showed that several system requirements hinged on the decision between a host-target system or a fully portable tool set and the decision between VMS and Unix as the host operating system. These requirements included the functions needed to provide a user friendly front end, the operating system to be used by the workstations, and the functions necessary to support a host-target operation. To keep these requirements in synchronization with the others, a special minispinal was initiated to address and resolve these issues. The resulting review led to a commitment to a host-target operation using Unix on the host system, at a point early enough to work the OS dependent requirements in a timely fashion.

- Addressing the risks of mismatches to the user-project's needs and priorities resulted in substantial participation of the user-project personnel in the requirements definition activity. This led to several significant redirections of the requirements, particularly toward supporting the early phases of the software life cycle into which the user project was embarking, such as an adaptation of the software requirements engineering methodology (SREM) tools for requirements specification and analysis.

It is also interesting to note that the form of Tables 1, 2, and 3 was originally developed for presentation purposes, but subsequently became a standard "spiral model template" used on later projects. These templates are useful not only for organizing project activities, but also as a residual design-rationale record. Design rationale information is of paramount importance in assessing the potential reusability of software components on future projects. Another important point to note is that the use of the template was indeed uniform across the three cycles, showing that the spiral steps can be and were uniformly followed at successively detailed levels of product definition.

Succeeding rounds. It will be useful to illustrate some examples of how the spinal model is used to handle situations arising in the preliminary design and detailed design of components of the SPS: the preliminary design specification for the requirements traceability tool (RTT), and a detailed design rework on go-back on the unit development folder (UDF) tool.

The RTT preliminary design specification. The RTT establishes the traceability between itemized software requirements specifications, design elements, code elements, and test cases. It also supports various associated query, analysis, and report generation capabilities. The preliminary design specification for the RTT (and most of the other SPS tools) looks different from the usual preliminary design specification, which tends to show a uniform level of elaboration of all components of the design. Instead, the level of detail of the RTT specification is risk-driven.

In areas involving a high risk if the design turned out to be wrong, the design was carried down to the detailed design level, usually with the aid of rapid prototyping. These areas included working out the implications of "undo" options and dealing with the effects of control keys used to escape from various program levels.

In areas involving a moderate risk if the design was wrong, the design was carried down to a preliminary-design level. These areas included the basic command options for the tool and the schemata for the requirements traceability database. Here again, the ease of rapid prototyping with Unix shell scripts supported a good deal of user-interface prototyping.

In areas involving a low risk if the design was wrong, very little design elaboration was done. These areas included details of all the help message options and all the report-generation options, once the nature of these options was established in some example instances.

A detailed design go-back. The UDF tool collects into an electronic "folder" all artifacts involved in the development of a single-programmer software unit (typically 500 to 1,000 instructions): unit requirements, design, code, test cases, test results, and documentation. It also includes a management template for tracking the programmer's scheduled and actual completion of each artifact.

An alternative considered during detailed design of the UDF tool was reuse of portions of the RTT to provide pointers to the requirements and preliminary design

specifications of the unit being developed. This turned out to be an extremely attractive alter-native, not only for avoiding duplicate software development but also for bringing to the surface several issues involving many-to-many mappings between requirements, design, and code that had not been considered in designing the UDF tool. These led to a rethinking of the UDF tool requirements and preliminary design, which avoided a great deal of code rework that would have been necessary if the detailed design of the UDF tool had proceeded in a purely deductive, top-down fashion from the original UDF requirements specification. The resulting go-back led to a significantly different, less costly, and more capable UDF tool, incorporating the RTT in its "uses-hierarchy."

Spiral model features. These two examples illustrate several features of the spiral approach.

- It fosters the development of specifications that are not necessarily uniform, exhaustive, or formal, in that they defer detailed elaboration of low-risk software elements and avoid unnecessary breakage in their design until the high-risk elements of the design are stabilized.

- It incorporates prototyping as a risk reduction option at any stage of development. In fact, prototyping and reuse risk analyses were often used in the process of going from detailed design into code.

- It accommodates reworks on go-backs to earlier stages as more attractive alternatives are identified on as new risk issues need resolution.

Overall, risk-driven documents, particularly specifications and plans, are important features of the spinal model. Great amounts of detail are not necessary unless the absence of such detail jeopardizes the project. In some cases, such as with a product whose functionality may be determined by a choice among commercial products, a set of weighted evaluation criteria for the products may be preferable to a detailed pre-statement of functional requirements.

Results. The Software Productivity System developed and supported using the spiral model avoided the identified risks and achieved most of the system's objectives. The SPS has grown to include over 300 tools and over 1,300,000 instructions; 93 percent of the instructions were reused from previous project-developed, TRW-developed, or external-software packages. Over 25 projects have used all on portions of the system. All of the projects fully using the system have increased their productivity at least 50%; indeed, most have doubled their productivity (when compared with cost-estimation model predictions of their productivity using traditional methods).

However, one risk area—that projects with non-Unix target systems would not accept a Unix-based host system—was underestimated. Some projects accepted the host—target approach, but for various reasons (such as customer constraints and zero-cost target machines) a good many did not. As a result, the system was less widely used on TRW projects than expected. This and other lessons learned have been incorporated

into the spiral model approach to developing TRW's next-generation software development environment.

Evaluation

Advantages. The primary advantage of the spiral model is that its range of options accommodates the good features of existing software process models, while its risk-driven approach avoids many of their difficulties. In appropriate situations, the spiral model becomes equivalent to one of the existing process models. In other situations, it provides guidance on the best mix of existing approaches to a given project; for example, its application to the TRW-SPS provided a risk-driven mix of specifying, prototyping, and evolutionary development.

The primary conditions under which the spiral model becomes equivalent to other main process models are summarized as follows:

- If a project has a low risk in such areas as getting the wrong user interface or not meeting stringent performance requirements, and if it has a high risk in budget and schedule predictability and control, then these risk considerations drive the spiral model into an equivalence to the waterfall model.

- If a software product's requirements are very stable (implying a low risk of expensive design and code breakage due to requirements changes during development), and if the presence of errors in the software product constitutes a high risk to the mission it serves, then these risk considerations drive the spiral model to resemble the two-leg model of precise specification and formal deductive program development.

- If a project has a low risk in such areas as losing budget and schedule predictability and control, encountering large-system integration problems, or coping with information sclerosis, and if it has a high risk in such areas as getting the wrong user interface or user decision support requirements, then these risk considerations drive the spinal model into an equivalence to the evolutionary development model.

- If automated software generation capabilities are available, then the spiral model accommodates them either as options for rapid prototyping on for application of the transform model, depending on the risk considerations involved.

- If the high-risk elements of a project involve a mix of the risk items listed above, then the spinal approach will reflect an appropriate mix of the process models above (as exemplified in the TRW-SPS application). In doing so, its risk avoidance features will generally avoid the difficulties of the other models.

The spiral model has a number of additional advantages, summarized as follows:

It focuses early attention on options involving the reuse of existing software. The steps involving the identification and evaluation of alternatives encourage these options.

It accommodates preparation for life-cycle evolution, growth, and changes of the software product. The major sources of product change are included in the prod-uct's objectives, and information-hiding approaches are attractive architectural design alternatives in that they reduce the risk of not being able to accommodate the product-charge objectives.

It provides a mechanism for incorporating software quality objectives into software product development. This mechanism derives from the emphasis on identifying all types of objectives and constraints during each round of the spiral. For example, Table 3 shows user-friendliness, portability, and reliability as specific objectives and constraints to be addressed by the SPS. In Table 1, security constraints were identified as a key risk item for the SPS.

It focuses on eliminating errors and unattractive alternatives early. The risk analysis, validation, and commitment steps cover these considerations.

For each of the sources of project activity and resource expenditure, it answers the key question, "How much is enough?" Stated another way, "How much of requirements analysis, planning, configuration management, quality assurance, testing, formal verification, and so on should a project do?" Using the risk-driven approach, one can see that the answer is not the same for all projects and that the appropriate level of effort is determined by the level of risk incurred by not doing enough.

It does not involve separate approaches for software development and software enhancement (or maintenance). This aspect helps avoid the "second-class citizen" status frequently associated with software maintenance. It also helps avoid many of the problems that currently ensue when high-risk enhancement efforts are approached in the same way as routine maintenance efforts.

It provides a viable framework for integrated hardware-software system development. The focus on risk management and on eliminating unattractive alternatives early and inexpensively is equally applicable to hardware and software.

Difficulties. The full spiral model can be successfully applied in many situations, but some difficulties must be addressed before it can be called a mature, universally applicable model. The three primary challenges involve matching to contract software, relying on risk-assessment expertise, and the need for further elaboration of spiral model steps.

Matching to contract software. The spiral model currently works well on internal software developments like the TRW-SPS, but it needs further work to match it to the world of contract software acquisition.

Internal software developments have a great deal of flexibility and freedom to accommodate stage-by-stage commitments, to defer commitments to specific options, to establish minispinals to resolve critical-path items, to adjust levels of effort, or to accommodate such practices as prototyping, evolutionary development, or design-to-cost. The world of contract software acquisition has a harder time achieving these degrees of flexibility and freedom without losing accountability and control, and a harder time defining contracts whose deliverables are not well specified in advance.

Recently, a good deal of progress has been made in establishing more flexible contract mechanisms, such as the use of competitive front-end contracts for concept definition or prototype fly-offs, the use of level-of-effort and award-fee contracts for evolutionary development, and the use of design-to-cost contracts. Although these have been generally successful, the procedures for using them still need to be worked out to the point that acquisition managers feel fully comfortable using them.

Relying on risk-assessment expertise. The spiral model places a great deal of reliance on the ability of software developers to identify and manage sources of project risk.

A good example of this is the spiral model's risk-driven specification, which carries high-risk elements down to a great deal of detail and leaves low-risk elements to be elaborated in later stages; by this time, there is less risk of breakage.

However, a team of inexperienced or low-balling developers may also produce a specification with a different pattern of variation in levels of detail: a great elaboration of detail for the well-understood, low-risk elements, and little elaboration of the poorly under stood, high-risk elements. Unless there is an insightful review of such a specification by experienced development or acquisition personnel, this type of project will give an illusion of progress during a period in which it is actually heading for disaster.

Another concern is that a risk-driven specification will also be people-dependent. For example, a design produced by an expert may be implemented by non-experts. In this case, the expert, who does not need a great deal of detailed documentation, must produce enough additional documentation to keep the non-experts from going astray. Reviewers of the specification must also be sensitive to these concerns.

With a conventional, document-driven approach, the requirement to carry all aspects of the specification to a uniform level of detail eliminates some potential problems and permits adequate review of some aspects by inexperienced reviewers. But it also creates a large drain on the time of the scarce experts, who must dig for the critical issues within a large mass of non-critical detail. Furthermore, if the high-risk elements have been glossed oven by impressive-sounding references to poorly understood capabilities (such as a new synchronization concept or a commercial DBMS), there is an even greater risk that the conventional approach will give the illusion of progress in situations that are actually heading for disaster.

Need for further elaboration of spiral model steps. In general, the spiral model process steps need further elaboration to ensure that all software development participants are operating in a consistent context.

Some examples of this are the need for more detailed definitions of the nature of spiral model specifications and milestones, the nature and objectives of spiral model reviews, techniques for estimating and synchronizing schedules, and the nature of spiral model status indicators and cost-versus-progress tracking procedures. Another need is for guidelines and checklists to identify the most likely sources of project risk and the most effective risk-resolution techniques for each source of risk.

Highly experienced people can successfully use the spiral approach without these elaborations. However, for large-scale use in situations in which people bring widely differing experience bases to the project, added levels of elaboration—such as have been accumulated oven the years for document-driven approaches—are important in ensuring consistent interpretation and use of the spinal approach across the project.

Efforts to apply and refine the spiral model have focused on creating a discipline of software risk management, including techniques for risk identification, risk analysis, risk prioritization, risk management planning, and risk-element tracking. The prioritized top-ten list of software risk items given in Table 4 is one result of this activity. Another exam-pie is the risk management plan discussed in the next section.

Table 4. A prioritized top-ten list of software risk items

Risk Item	Risk management techniques
1. Personnel shortfalls	Staffing with top talent, job matching; teambuilding; morale building; cross-training; pre-scheduling key people
2. Unrealistic schedules and budgets	Detailed, multisource cost and schedule estimation; design to cost; incremental development; software reuse; requirements scrubbing
3. Developing the wrong software functions	Organization analysis; mission analysis; ops-concept formulation; user surveys; prototyping; early users' manuals
4. Developing the wrong user interface	Task analysis; prototyping; scenarios; user characterization (functionality, style, workload)
5. Gold plating	Requirements scrubbing; prototyping; cost-benefit analysis; design to cost
6. Continuing stream of requirement changes	High change threshold; information hiding; incremental development (defer changes to later increments)
7. Shortfalls in externally furnished components	Benchmarking; inspections; reference checking; compatibility analysis

8.	Shortfalls in externally performed tasks	Reference checking; pre-award audits; award-fee contracts; competitive design or prototyping; teambuilding
9.	Real-time performance shortfalls	Simulation; benchmarking; modeling; prototyping; instrumentation; tuning
10.	Straining computer-science capabilities	Technical analysis; cost—benefit analysis; prototyping; reference checking

Implications: The Risk Management Plan. Even if an organization is not ready to adopt the entire spiral approach, one characteristic technique that can easily be adapted to any life-cycle model provides many of the benefits of the spiral approach. This is the risk management plan summarized in Table 5. This plan basically ensures that each project makes an early identification of its top risk items (the number 10 is not an absolute requirement), develops a strategy for resolving the risk items, identifies and sets down an agenda to resolve new risk items as they surface, and highlights progress versus plans in monthly reviews.

Table 5. Software risk management plan

1.	Identify the project's top 10 risk items.
2.	Present a plan for resolving each risk item.
3.	Update list of top risk items, plan, and results monthly.
4.	Highlight risk-item status in monthly project reviews. • Compare with previous month's rankings, status.
5.	Initiate appropriate corrective actions.

The risk management plan has been used successfully at TRW and other organizations. Its use has ensured appropriate focus on early prototyping, simulation, benchmarking, key-person staffing measures, and other early risk-resolution techniques that have helped avoid many potential project "show-stoppers." The recent US Department of Defense standard on software management, DoD-Std-2167, requires that developers produce and use risk management plans, as does its counterpart US Air Force regulation, AFR 800-14.

Overall, the Risk Management Plan and the maturing set of techniques for software risk management provide a foundation for tailoring spinal model concepts into the more established software acquisition and development procedures.

We can draw four conclusions from the data presented:

(1) The risk-driven nature of the spiral model is more adaptable to the full range of software project situations than are the primarily document-driven approaches such as the waterfall model or the primarily code-driven approaches such as evolutionary development. It is particularly applicable to very large, complex, ambitious soft-wane systems.

(2) The spiral model has been quite successful in its largest application to date: the development and enhancement of the TRW-SPS. Overall, it achieved a high level of software support environment capability in a very short time and provided the flexibility necessary to accommodate a high dynamic range of technical alternatives and user objectives.

(3) The spiral model is not yet as fully elaborated as the more established models. Therefore, the spinal model can be applied by experienced personnel, but it needs further elaboration in such areas as contracting, specifications, milestones, reviews, scheduling, status monitoring, and risk area identification to be fully usable in all situations.

(4) Partial implementations of the spiral model, such as the risk management plan, are compatible with most current process models and are very helpful in overcoming major sources of project risk.

Acknowledgments

I would like to thank Frank Belz, Lob Penedo, George Spadano, Bob Williams, Bob Balzen, Gillian Frewin, Peter Hamer, Manny Lehman, Lee Ostenweil, Dave Parnas, Bill Riddle, Steve Squires, and Dick Thayen, along with the Computer reviewers of this article, for their stimulating and insightful comments and discussions of earlier versions, and Nancy Donato for producing its several versions.

References

1. F.P. Brooks et al., *Defense Science Board Task Force Report on Military Software*, Office of the Under Secretary of Defense for Acquisition, Washington, DC 20301, Sept. 1987.

2. H.D. Benington, “Production of Large Computer Programs,” *Proc. ONR Symp. Advanced Programming Methods for Digital Computers*, June 1956, pp. 15–27. Also available in *Annals of the History of Computing*, Oct. 1983, pp. 350–361, and *Proc. Ninth Int’l Conf Software Engineering*, Computer Society Press, 1987.

3. W.W. Royce, “Managing the Development of Large Software Systems: Concepts and Techniques,” *Proc. Wescon*, Aug. 1970. Also available in *Proc. ICSE 9*, Computer Society Press, 1987.

4. D.D. McCracken and M.A. Jackson, “Life-Cycle Concept Considered Harmful,” *ACM Software Engineering Notes*, Apr. 1982, pp. 29–32.

5. R. Balzer, T.E. Cheatham, and C. Green, “Software Technology in the I990s: Using a New Paradigm,” *Computer*, Nov. 1983, pp. 39–45.

6. B.W. Boehm et al., “A Software Development Environment for Improving Productivity,” *Computer*, June 1984, pp. 30–44.

7. B.W. Boehm, *Software Engineering Economics*, Prentice-Hall, 1981, Chap. 33.

Further reading

The software process model field has an interesting history, and a great deal of stimulating work has been produced recently in this specialized area. Besides the references to this article, here are some additional good sources of insight:

Overall process model issues and results

Agresti’s tutorial volume provides a good overview and set of key articles. The three recent *Software Process Workshop Proceedings* provide access to much of the recent work in the area.

Agresti, W.W., *New Paradigms for Software Development*, IEEE Catalog No. EH0245-1, 1986.

Dowson, M., ed., *Proc. Third Int’l Software Process Workshop*, IEEE Catalog No. TH0184-2, Nov. 1986.

Potts, C., ed., *Proc. Software Process Workshop*, IEEE Catalog No. 84CH2044-6, Feb. 1984.

Wileden, J.C., and M. Dowson, eds., Proc. Int’l Workshop Software Process and Software Environments, *ACM Software Engineering Notes*, Aug. 1986.

Alternative process models

More detailed information on waterfall-type approaches is given in:

Evans, M.W., P. Piazza, and J.P. Dolkas, *Principles of Productive Software Management*, John Wiley & Sons, 1983.

Hice, G.F., W.J. Turner, and L.F. Cashwell, *System Development Methodology*, North Holland, 1974 (2nd ed., 1981).

More detailed information on evolutionary development is provided in:

Gilb, T., *Principles of Software Engineering Management*, Addison-Wesley, 1988 (currently in publication).

Some additional process model approaches with useful features and insights may be found in:

Lehman, M.M., and L.A. Belady, *Program Evolution: Processes of Software Change*, Academic Press, 1985.

Osterweil, L., "Software Processes are Software, Too," *Proc. ICSE 9*, IEEE Catalog No. 87CH2432-3, Mar. 1987, pp. 2–13.

Radice, R.A., et al., "A Programming Process Architecture," *IBM Systems J.*, Vol. 24, No. 2, 1985, pp. 79–90.

Spiral and spiral-type models

Some further treatments of spiral model issues and practices are:

Belz, F.C., "Applying the Spiral Model: Observations on Developing System Software in Ada," *Proc. 1986 Annual Conf. on Ada Technology*, Atlanta, 1986, pp. 57–66.

Boehm, B.W., and F.C. Belz, "Applying Process Programming to the Spiral Model," *Proc. Fourth Software Process Workshop*, IEEE, May 1988.

Iivari, J., "A Hierarchical Spiral Model for the Software Process," *ACM Software Engineering Notes*, Jan. 1987, pp. 33–37.

Some similar cyclic spinal-type process models from other fields are described in:

Carlsson, B., P. Keane, and J.B. Martin, "R&D Organizations as Learning Systems," *Sloan Management Review*, Spring 1976, pp. 1–15.

Fisher, R., and W. Ury, *Getting to Yes*, Houghton Mifflin, 1981; Penguin Books, 1983, pp. 68–71.

Kolb, D.A., "On Management and the Learning Process," MIT Sloan School Working Article 652-73, Cambridge, Mass., 1973.

Software risk management

The discipline of software risk management provides a bridge between spiral model concepts and currently established software acquisition and development procedures.

Boehm, B.W., “Software Risk Management Tutorial,” Computer Society, Apr. 1988.

Risk Assessment Techniques, Defense Systems Management College, Ft. Belvoir, Va. 22060, July 1983.

Barry W. Boehm is the chief scientist of the TRW Defense Systems Group. Since 1973, he has been responsible for developing TRW’s software technology base. His current primary responsibilities are in the areas of software environments, process models, management methods, Ada, and cost estimation. He is also an adjunct professor at UCLA.

Boehm received his BA degree in mathematics from Harvard in 1957 and his MA and PhD from UCLA in 1961 and 1964, respectively.