III. SOFTWARE PROCESS MODELS

Session Summary

The topic of the second session was models of the software process. Barry Boehm (TRW) outlined a spiral model of software development and enhancement in his keynote presentation. This model describes the software process as an iteration on four phases of activity, and could, Barry suggested, be particularized to fit with a variety of approaches and methods. Tom Cheatham (Harvard) discussed the knowledge-based software assistant (KBSA) paradigm, focusing on the role of the KBSA "activity coordinator" in mediating the various facets of software development activity. Some debate ensued over the extent to which the transformations envisioned in the KBSA paradigm were "magical". Other questions were raised about both speakers' models, including suggestions that each of them was more appropriately seen as a metamodel. There did seem to be some emerging consensus that process models have some inherently cyclic nature.

This section of the proceedings includes a paper based on Boehm's keynote presentation and position statements relevant to the session topic submitted by Bruce Blum and Mark Dowson. Material related to the topic of Cheatham's keynote presentation may be found in a paper by Balzer, Cheatham and Green that appeared in IEEE Computer in November 1983.

A SPIRAL MODEL OF SOFTWARE DEVELOPMENT AND ENHANCEMENT

by

Barry W. Boehm

TRW Defense Systems Group

1. Introduction

1.1. Overview

The spiral model of software development and enhancement presented here provides a new framework for guiding the software process. Its major distinguishing feature is that it creates a *risk-driven* approach to the software process, rather than a strictly specification-driven or prototype-driven process. It incorporates many of the strengths of other models, while resolving many of their difficulties.

This section presents a short historical background of software process models and the issues they address. Section 2 summarizes the process steps involved in the spiral model. Section 3 illustrates the application of the spiral model to a software project, using the TRW Software Productivity Project as an example. Section 4 summarizes the primary advantages, challenges, and implications involved in using the spiral model, and Section 5 presents the resulting conclusions.

1.2. Background

The Waterfall Model

One of the earliest software process models is the *stagewise* model given in [Bennington, 1956]. This model recommends that software be developed in successive stages (operational plan, operational specifications, coding specifications, coding, parameter testing, assembly testing, shakedown, system evaluation).

The original treatment of the *waterfall model* given in [Royce, 1970], etc. provided two primary enhancements to the stagewise model:

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

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

The waterfall approach was largely consistent with the *top-down structured programming model* introduced in [Mills, 1971]. However, some attempts to apply these versions of the waterfall model ran into the following kinds of difficulties:

- The "build it twice" step was unnecessary in some situations (e.g., developing a well-understood payroll system), and subject to such unproductive phenomena as the "second system syndrome" [Brooks, 1975].

- The pure top-down approach needed to be tempered with a "look ahead" step to cover such issues as high-risk, low-level elements and reusable or common software modules.

These considerations resulted in the *risk-management variant* of the waterfall model discussed in [Boehm, 1975] and elaborated in [Boehm, 1976]. In this variant, each step was expanded to include a validation and verification activity to cover high-risk elements, reuse considerations, and prototyping. Further elaborations of the waterfall model covered such practices as incremental development [Distaso, 1980].

Alternative Software Life-Cycle Models

Still, the waterfall model continued to encounter a number of difficulties. These have led to a number of alternative life-cycle models which do a better job of coping with the difficulties:

1. The waterfall model does not adequately address concerns of developing program families and organising software to accommodate change. The Parnas *information-hiding approach* [Parnas, 1979] does an excellent job of addressing these concerns.

2. The waterfall model assumes a relatively uniform progression of elaboration steps. The "two-leg" model [Lehman-Stenning-Turski, 1984], [Lehman, 1984] features separate processes of abstraction until a formal specification is achieved, followed by a set of formal deductive "reification" steps to proceed through design and into code.

3. The waterfall model does not accommodate the sort of evolutionary development made possible by rapid prototyping capabilities and fourth-generation languages. Several *evolutionary development* models [McCracken-Jackson, 1982] and mixed models [Giddings, 1984] have been advanced to address this approach.

4. The waterfall model does not address the possible future modes of software development associated with automatic programming capabilities, program transformation capabilities, and "knowledge-based software assistant" capabilities. The automation paradigm [Balser-Cheatham-Green, 1983] provides an alternative life-cycle model and conceptual framework for incorporating these capabilities.

However, although each of these alternative approaches deals with some of the difficulties of the

waterfall approach, each has its own set of difficulties and challenges to resolve. The information-hiding approach has not yet been fully elaborated to see how it will cover such issues as prototyping and reuse of previous software. The two-leg model has challenges in accommodating software reuse, program families, and logical-physical design tradeoffs. The evolutionary development approach has challenges in scaling up to very large systems, ensuring process visibility and control, avoiding the negative effects of "information sclerosis;"[1] and avoiding the "undisciplined hacker" approach that the waterfall and other models were trying to correct. The automation paradigm has challenges in scaling up to very large systems accommodating program families, avoiding the effects of information sclerosis, and handling choices between older, stable, but less powerful capabilities and new, unstable, but more powerful capabilities. We will now proceed to describe the spiral model and show how it addresses these difficulties and challenges.

2. The Spiral Model

The spiral model of the software process has been evolving at TRW for several years, based on experience with various refinements of the waterfall model as applied to large government projects. The spiral model serves as a significantly more robust foundation for a software development environment than previous models; it includes most previous models as special cases, and further provides guidance as to which combination of previous models best fits a given software situation. Its most complete application to date has been to the development of the TRW Software Productivity System; this application will be described in Section 3.

The spiral model is illustrated in Figure 1. The radial dimension in Figure 1 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 holds that each cycle involves a progression through 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).

2.1. 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).

[1] Information sclerosis is a syndrome familiar to operational information-based systems, in which temporary work-arounds for software deficiencies increasingly solidify into unchangeable constraints on evolution. A typical example is the following comment: "It's nice that you could change those equipment codes to make them more intelligible for us, but the Codes Committee just met and established the current codes as company standards."

PIRAL MODEL OF THE SOFTWARE PROCESS

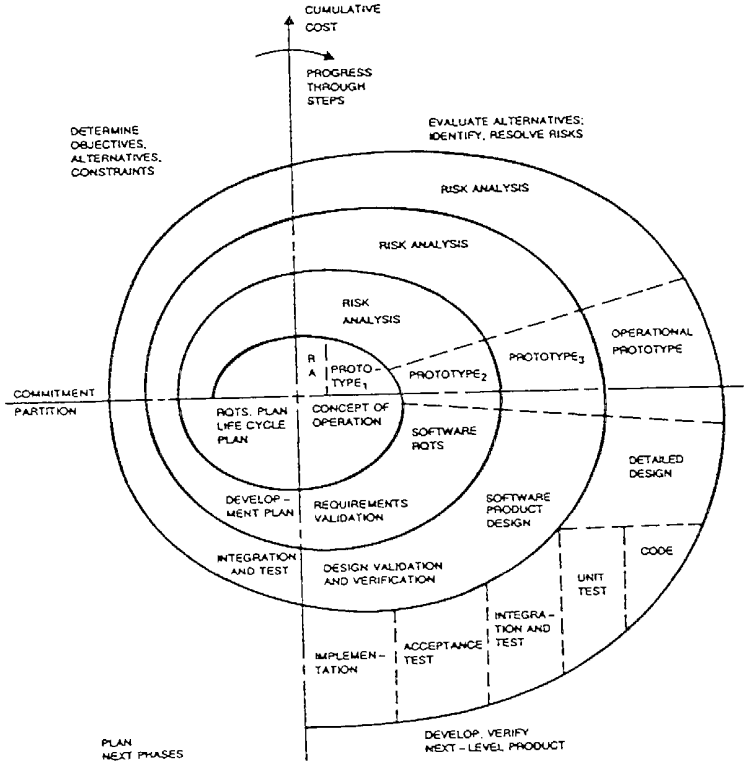

Figure 1 : Spiral Model of the Software Process

- The constraints imposed on the application of the alternatives (cost, schedule, interface, etc.).

The next step is to evaluate the alternatives with respect to the objectives and constraints. Frequently, this process will identify areas of uncertainty which 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, 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 risks remaining. If performance or user-interface risks strongly dominate program development or internal interface-control risks, the next step may be an evolutionary development step: 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 major risk issues. 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, modified as appropriate to incorporate incremental development.

The spiral model also accommodates any appropriate mixture of specification-oriented, prototype-oriented, simulation-oriented, automatic transformation-oriented, or other approaches to software development, where 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 determine the amount of time and effort which should be devoted to such other project activities as planning, configuration management, quality assurance, formal verification, or testing).

An important feature of the spiral model is that each cycle is completed by a review involving the primary people or organizations concerned with the product. This review covers all of the products developed during the previous cycle, including the plans for the next cycle and the resources required to carry them out. The major objective of the review is to ensure that all concerned parties are mutually committed to the approach to be taken 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. Thus, the review and commitment step may range from an individual walkthrough of the design of a single programmer component, to a major requirements review involving developer, customer, user, and maintenance organizations.

2.2. Initiating the Spiral: Mission Opportunity Model

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 answer to these questions involves an observation that the spiral model applies equally well to development or enhancement efforts, each of which are initiated 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, the spiral is terminated. Otherwise, it terminates in the installation of the new or modified software, and the hypothesis is tested by observing the effect on the operational mission.

These aspects of using the spiral model are covered by the complementary *Mission Opportunity* Model shown in Figure 2. The Mission Opportunity Model indicates that any software development or enhancement activity results from a decision based on observing and understanding an operational mission or set of missions, and concluding that a software effort would improve the cost-effectiveness of the mission operations. This decision serves to initiate the spiral model, which then may consist of very elaborate rounds of system definition and development, or very rapid rounds converging on the best way to implement a small enhancement.

3. Using the Spiral Model: The TRW Software Productivity System

The various rounds and activities involved in the spiral model are best understood via an example. This section will show how the spiral model was used in the definition and development of the TRW Software Productivity System (SPS) an integrated software engineering environment described in [Boehm et al, 1984]. 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 a potentially attractive area to investigate. This initially led to a small, extra "Round 0" circuit of the spiral to determine whether it was feasible to significantly increase software productivity at a reasonable corporate cost[2].

[2] Very large or complex software projects will frequently precede the "concept of operation" round of the spiral with one or more smaller rounds to establish feasibility and to reduce the range of alternative solutions quickly and inexpensively.

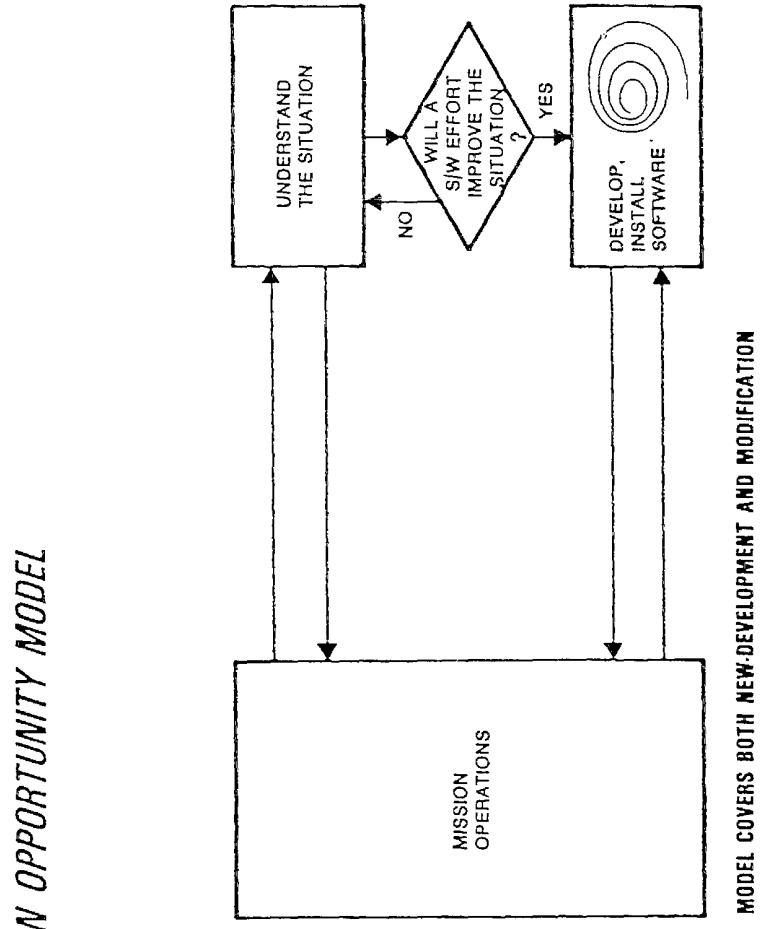

Figure 2 : The Mission Opportunity Model

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

3.1. Round 0: Feasibility Study

This study involved five part-time participants over roughly a 2-3 month time span. As indicated in Table 1, the objectives and constraints for this round are expressed at a very high level, and in more qualitative terms: "significantly increase," "at reasonable cost", etc.

Round 0 considered a wide number of alternative approaches to significantly improve software productivity at a reasonable cost. Some of these, primarily in the "technology" area, could lead to the development of a software product, but there were also a number of alternatives in the management, personnel, and facilities areas which could have led to a conclusion not to embark on a software development activity.

The primary risk areas needing resolution were the possibility of situations in which the company would invest a good deal into improving software productivity and find that:

- The resulting productivity gains would not be significant; or that
- Any potentially high-leverage improvements would be incompatible 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 COCOMO software cost model (see [Boehm, 1981], Chapter 33); and an analysis of previous projects at TRW exhibiting high levels of productivity.

The risk analysis results indicated that it was highly likely 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 a level of commitment on the order of 12 man-months as compared to the 2 man-months invested in Round 0. Round 1 here corresponded fairly well with the initial round of the spiral model shown in Figure 1, in that its intent was to produce a concept of operation and a basic

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 and 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 • Analyse Cost Model • Analyse Exceptional Projects • Literature Search
Risk Resolution Results	• Some Alternatives Infeasible - Single Time Sharing System : Security • Mix of Alternatives Can Produce Significant Gains - Factor of 2 in 5 years • Need Further Study to Determine Best Mix
Plan for Next Phase	• 6-Person Task Force for 6 Months • More Extensive Surveys & Analysis - Internal, External, Economic • Develop Concept of Operation, Economic Rationale
Commitment	• Fund Next Phase

Table 1 : Spiral Model Usage : TRW Software Productivity System, Round 0

life-cycle plan for implementing whatever preferred alternative emerged.

3.2. Round 1: Concept of Operations

Table 2 provides a summary of Round 1 of the spiral along the same lines given in Table 1 for Round 0. Rather than simply elaborate on the entries in Table 2, the discussion below focuses on comparing and contrasting the features of Rounds 0 and 1:

- As mentioned above, the level of investment was greater (12 man-months vs. 2).
- The objectives and constraints were more specific ("double software productivity in 5 years at a cost of $10K/person" vs. "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 or SADT, etc." vs "tools"; "private or shared terminals, smart or dumb terminals" vs "workstations").
- The risk areas identified were more specific ("TRW LAN price-performance within a $10K/person investment constraint" vs "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 offices 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 a prototype software support environment (SDE).
- The commitment step involved more than just an agreement with the plan. It added a commitment for an upcoming 100-person software project to be the initial testbed user of the system, and for the environment development to focus on the needs of the testbed project. It also added the formation of 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 the development of a prototype environment, it also recommended that the project also employ requirements specification and design specifications in a risk-driven way. Thus, the development of the prototype followed the succeeding rounds of the spiral model.

Objectives	• Double Software Productivity in 5 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	• Ops. Concept : Private Offices, TRW LAN, Personal Terminals, VAX • Begin with Primarily Dumb Terminals; Experiment with Smart Workstations • Defer OS, Tools Selection
Plan for Next Phase	• Partition Effort into SDE, Facilities, Management • Develop First-Cut Prototype SDE - Design-to-Cost : 15 Person Team for 1 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

Table 2 : Spiral Model Usage : TRW Software Productivity System, Round 1

3.3. Round 2: Top-Level Requirements Specification

Table 3 shows the corresponding steps involved during Round 2 in defining the SPS. Since a number of these Round 2 decisions and their rationale are covered in [Boehm et al, 1984], we will not elaborate on them here. Instead, we will summarize two of the highlights dealing with risk management and the use of the spiral model.

- The initial risk-identification activities during Round 2 showed that several system requirements depended 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 dependent requirements included the functions required to provide a user-friendly front-end, the operating system to be used by the workstations, and the functions required to support a host-target operation. In order to keep these requirements in synchronization with the other requirements, a special mini-spiral was initiated to address and resolve these issues. The resulting review resulted in 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 in directing the early phase of the software life-cycle into which the user project was embarking, such as an adaptation of the Software Requirements Engineering Methodology (SREM) Tools [Alford, 1977].

3.4. Succeeding Rounds of the Spiral

Within the confines of this paper, it is not possible to discuss each round of the spiral in detail. But it will be useful to illustrate some examples of how the spiral 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 go-back on the Unit Development Folder (UDF) tool.

The RTT Preliminary Design Specification

The preliminary design specification for the RTT (and most of the other SPS tools) looked 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 was risk-driven:

- In areas involving a high risk if the design was 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 various "undo" options, and the effects of various control keys used to escape from various levels of the program.

- 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 data base. Here again, the ease of rapid prototyping with Unix shell scripts supported a good deal of user-interface prototyping.

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 Toolset • 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, OA Tools - for Front End, Support Tools - for LAN, Equipment, Facilities
Commitment	• Proceed with Plans

Table 5 : Spiral Model Usage : TRW Software Productivity System, Round 2

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

A Detailed Design Go-Back: The UDF Tool

During the detailed design of the Unit Development Folder (UDF) tool, an alternative was considered to reuse portions of the Requirements Traceability Tool (RTT) to provide pointers to the requirements and preliminary design specifications of the unit being developed. This turned out to be an extremely attractive alternative, not only in avoiding duplicate software development, but also in surfacing several issues involving many-to-many mappings between requirements, preliminary design, and detailed design which 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 and more capable UDF tool, incorporating the RTT in its "uses hierarchy."

Spiral Model Features Illustrated by the Two Examples

From these two examples, we can see that the spiral approach:

- 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 stablized.
- Incorporates prototyping as a risk-reduction option at any stage of development.
- Accommodates go-backs to earlier stages of the spiral as more attractive alternatives are identified or as new risk issues need resolution.

4. Spiral Model Advantages, Challenges, and Implications

4.1. Spiral Model Advantages

The primary advantage of the spiral model is that its range of options and risk-driven approach allow it to accommodate the best features of existing software process models, while avoiding most 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 be applied to a given project. The application of the spiral model to the TRW SPS discussed in Section 3 provides a good example of 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 summarised below.

- If a project has a low risk in such areas as getting the wrong user interface or not meeting stringent performance requirements; and it has a high risk if it loses budget, schedule, and product 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 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 spiral model into an equivalence to the evolutionary development model.
- If automated software development capabilities are available, then the spiral model accommodates them either as options for rapid prototyping or for application of the automation paradigm, depending on the risk considerations involved.
- If the high-risk elements of a project involve a mix of the risk items above, then the spiral approach will reflect an appropriate mix of the process models above. In doing so, its risk-avoidance features will generally avoid the difficulties of the other models.

In addition, the spiral model has a number of further advantages which are summarised below.

- *It accommodates strategies for developing program families, and for the reuse of existing software.* The steps involving the identification and evaluation of alternatives accommodate 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 product's objectives, and information-hiding approaches are included in the architectural design alternatives.
- *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. The GOALS approach and software engineering goal structure in [Boehm, 1981; Chapter 3] provide a process and checklist for incorporating quality objectives.
- *It focuses on eliminating errors and unattractive alternatives early.* The risk-analysis, validation, and commitment steps cover these considerations.
- *It accommodates iterations, go-backs, and early termination of non-viable software projects.* The first two of these aspects were illustrated in the TRW-SPS example. The example also illustrated the overall objective of the spiral approach to start small, keep the spiral as tight as possible, and thus achieve the project's objectives with a minimum resource expenditure.
- *For each of the sources of project activity and resource expenditure, it answers the key*

question, "how much is enough?" How much should a project do of requirements analysis, planning, configuration management, quality assurance, testing, formal verification, etc? Using the risk-driven approach, we 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 can support, and be supported by, advanced software development environments.* The process steps and their associated internal and external products can be treated as data base objects to be handled by an advanced object manager. Also, process and risk-management guidance for software developers can be incorporated into an evolving KBSA or activity coordinator.

- *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.

- *On the other hand, it avoids forcing software development procedures into hardware development paradigms.* A good example of this is the review called "Critical Design Review," or CDR, which occurs after the detailed design is completed. For a hardware development risk profile, this review is indeed critical: as seen in Figure 3, it is the final review before the project begins to commit the bulk of its resources into producing hardware. From this standpoint, it is indeed critical to invest in an exhaustive, across-the-board review of the detailed design specifications before proceeding. For a software project, however, the risk profile is significantly different. Typically, in order to reduce the crucial factor of software development time, a software project will work out detailed unit interfaces by the Preliminary Design Review (PDR), and then begin to commit the bulk of its resources to large numbers of people doing detailed design and code in parallel (see Figure 3). To interrupt this process with a single, large, across-the-board CDR is not only time-consuming and expensive, but also ineffective, as the inter-unit interface specifications were necessarily verified thoroughly at the PDR. Thus, the critical review for a software project is the PDR; the "detailed design review" function is performed much more cost-effectively by a program of individual design inspections or walkthroughs.

4.2. Spiral Model Challenges

Although the full spiral model can be successfully applied in many situations, it still has some challenges to address before it can be called a mature, universally applicable model. The three primary spiral model challenges are summarized below.

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-

RISK-ORIENTED MANAGEMENT REFOCUSES SOFTWARE REVIEWS ONTO KEY SOFTWARE ISSUES

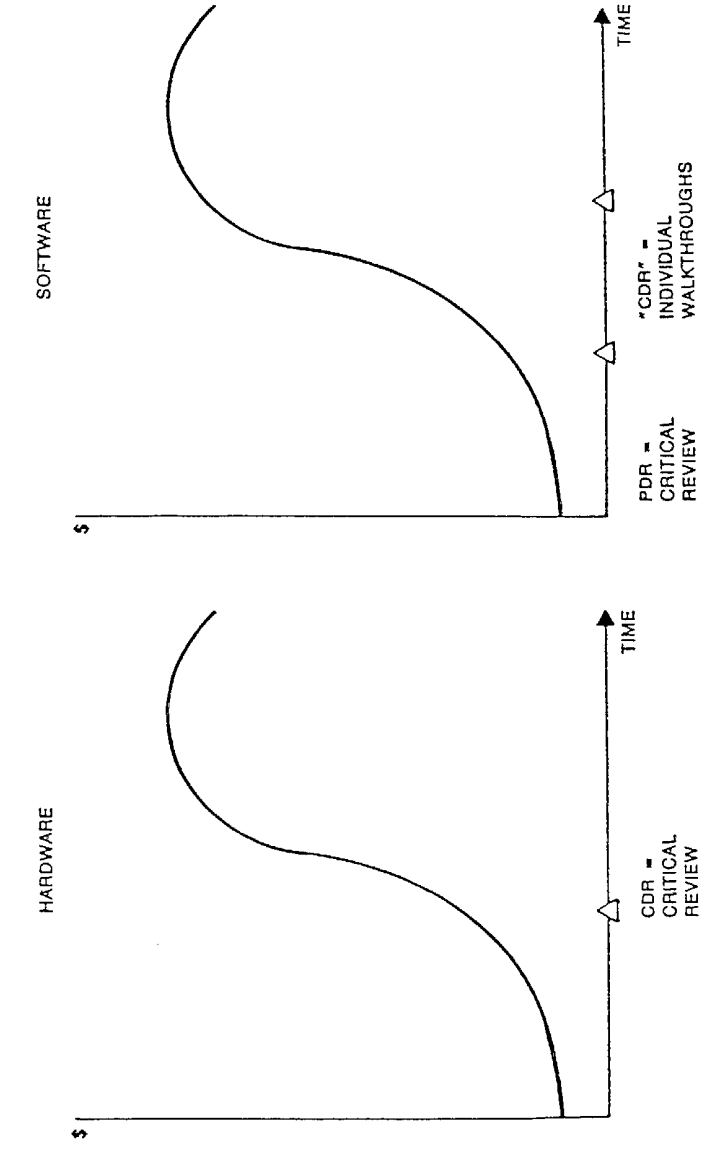

Figure 3 : Risk-Oriented Management Refocuses Software Reviews onto Key Software Issues

stage commitments, to defer commitments to specific options, to establish mini-spirals 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.

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 [AFSAB, 1983], the procedures for using them still need to be worked out to the point that acquisition managers feel fully comfortable in using them.

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 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 which time there is less risk of breakage.

However, a team of inexperienced or low-balling developers may produce a specification with a more dangerous 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-understood, high-risk elements. Unless there is an insightful review of such a specification by experienced developer or acquisition personnel, this type of project will proceed to 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 may be produced by an expert, but may be implemented by non-experts. In this case, even though the expert does not need a great deal of detailed documentation, it is necessary for him to produce enough additional documentation to ensure that the non-experts will not go astray. Reviewers of the specification must be sensitive to these concerns as well.

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 creates a situation in which some aspects of reviews can be adequately carried out 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. And further, if the high-risk elements have been glossed over by impressive-sounding references to poorly-understood capabilities (e.g., a new synchronization concept or a commercial DBMS), then there is an even greater risk of giving an illusion of progress in situations which are actually heading

for disaster.

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

Some examples are the need for more detailed definitions of the nature of spiral model specifications and milestones, the nature and objectives of spiral model reviews, and the nature of spiral model status indicators and cost-vs.-progress tracking procedures. Another need is for guidelines and checklists to be used in identifying the most likely sources of project risk, and the most effective risk-resolution techniques for each source of risk.

It is currently feasible for highly experienced people to successfully use the spiral approach without these elaborations. But for large scale use in situations where people bring widely differing experience bases to the project, added levels of elaboration – such as have been accumulated over the years for document-driven approaches – would be important in ensuring consistent interpretation and usage of the spiral approach across the project.

4.3. Spiral Model Implications: The Risk Management Plan

Even if an organization is not fully ready to adopt the entire spiral approach, there is one characteristic spiral model technique which can easily be adapted to any life-cycle model, and which an provide many of the benefits of the spiral approach. This is the *Risk Management Plan* summarized in Table 4. The Risk Management Plan basically ensures that each project make an early identification of its top risk items (the number 10 is not an absolute requirement), develop a plan for resolving the risk items, identify and plan to resolve new risk items as they surface, and highlight the project's progress vs. plans in monthly reviews.

The Risk Management Plan has been used successfully at TRW and other organizations, and has ensured an appropriate focus on early prototyping, simulation, benchmarking, key-person staffing measures, and other early risk-resolution techniques which have helped avoid many potential project "show-stoppers." The recent U.S. Air Force Summer Study on Software Acquisition [AFSAB, 1983] recommended that the Air Force adopt the Risk Management Plan as standard practice, and efforts in this direction are underway.

5. Conclusions

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 software systems.

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
5. Initiate Appropriate Corrective Actions

Table 4 : Software Risk Management Plan

2. The spiral model has been quite successful in its largest application to date: the development and enhancement of the TRW SPS. Overall, the spiral approach was highly effective in achieving a high level of software support environment capability in a very short time, and in providing the flexibility necessary to accommodate a wide and dynamic range of technical alternatives and user objectives.

3. The spiral model is not yet as fully elaborated as are more established models such as the waterfall model. Thus, although the spiral model can be successfully applied by experienced personnel, it needs further elaboration in such areas as contracting, specifications, milestones, reviews, status monitoring, and risk-area identification in order 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 highly useful in helping to overcome their major sources of project risk.

REFERENCES

1. [AFSAB, 1983]. USAF Scientific Advisory Board, Report of the USAF/SAB Committee on the High Cost and Risk of Mission-Critical Software, J. B. Munson, chair, Dec. 1983.

2. [Alford, 1977] Alford, M. W., "A Requirements Engineering Methodology for Real-Time Processing Requirements," IEEE Trans. S/W Engr., JAN 1977, pp. 60-68.

3. [Balzer-Cheatham-Green, 1983] Balzer, R., T.E. Cheatham, and C. Green, "Software Technology in the 1990's: Using a New Paradigm," Computer, Nov. 1983, pp 39-45.

4. [Benington, 1956]. Benington, H. D., "Production of Large Computer Programs," Proc. ONR Symposium on 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.

5. [Boehm, 1975]. Boehm, B. W., " Software Design and Structuring," in Practical Strategies for Developing Large Software Systems, E. Horowitz (ed). Addison-Wesley, 1975, pp. 103-128.

6. [Boehm, 1976]. Boehm, B. W., "Software Engineering," IEEE Trans. Computers, December 1976, pp. 1226-1241.

7. [Boehm, 1981]. Boehm, B. W., Software Engineering Economics, Prentice- Hall, 1981.

8. [Boehm, et al, 1984]. Boehm, B. W., M.H. Penedo, E. D. Stuckle, R. D. Williams, and A. B. Pyster, "A Software Development Environment for Improving Productivity," Computer, June 1984, pp. 30-44.

9. [Brooks, 1975]. Brooks, F.P., The Mythical Man-Month, Addison- Wesley, 1975.

10. [Distaso, 1980]. Distaso, J. R., "Software Management- A Survey of the Practice in 1980." IEEE Proceedings, Sept. 1980, pp. 1103-1119.

11. [Giddings, 1984]. Giddings, R.V., "Accommodating Uncertainty in Software Design," Comm. ACM, May 1984, pp. 428-434.

12. [Lehman, 1984]. Lehman, M. M., "A Further Model of Coherent Programming Processes," Proceedings, Software Process Workshop, IEEE, Feb. 1984, pp. 27-33.

13. [Lehman-Stenning-Turski, 1984]. Lehman, M. M., V. Stenning, and W. Turski, "Another Look at Software Design Methodology," Software Engineering Notes, ACM, Apr. 1984, pp. 38-53.

14. [McCracken-Jackson, 1982]. D. D. McCracken and M. A. Jackson, "Life Cycle Concept Considered Harmful," Software Engineering Notes, ACM, April 1982, pp. 29-32.

15. [Mills, 1971]. Mills, H. D., "Top-Down Programming in Large Systems," in Debugging Techniques in Large Systems, R. Ruskin (ed), Prentice-Hall, 1971, pp. 41-55.

16. [Parnas, 1979]. Parnas, D.L., "Designing Software for Ease of Extension and Contraction," IEEE Trans. S/W. Engr., March 1979, pp. 128-137.

17. [Royce, 1970]. Royce, W. W., "Managing the Development of Large Software Systems: Concepts and Techniques," Proceedings, WESCON, August 1970.