Risk Associated with Software Development: A Holistic Framework for Assessment and Management

Clyde Chittister, *Member, IEEE*, and Yacov Y. Haimes, *Fellow, IEEE*

Abstract— **Although the management of risk in the development process is critical for all engineering disciplines, this paper focuses on the software development process and proposes a framework for the assessment and management of risk associated with this process. The proposed framework is grounded on a holistic concept termed hierarchical holographic modeling, where more than one perspective or vision of the risk associated with software development is analyzed. Three perspectives, or decompositions, are introduced: 1) functional decomposition, which encompasses seven basic attributes associated with software development—requirement, product, process, people, management, environment, and system development; 2) source-based decomposition, which relates to the four sources of failure—hardware, software, organizational, and human; and 3) temporal decomposition, which relates to the stages in the software development process. The following set of questions is addressed at each level of the hierarchical holographic submodels: What can go wrong? What is the likelihood that it will go wrong? What are the consequences? What can be done? What options are available? What are the associated trade-offs in terms of all costs, benefits, and risks? And what is the impact of current management decisions on future options? Once the "universe" of risk-based problems has been identified, then a risk ranking method is applied to provide priorities among them. Because software development is an intellectual, labor-intensive activity, this paper pays special attention to the role of human resource development and improvement in risk assessment. The paper is the first among a set of articles on the risk associated with software development.**

I. Introduction

COMPUTERS have become pervasive in our society. They are integral to everything from VCRs and video games to power plants and control systems for aircraft. Computers enhance satellite communications systems that provide television nationwide and enabled the military (as well as CNN) to communicate during the Iraq war. Computers touch the lives of the majority of Americans daily.

Computers are composed of two major components. One is hardware: the power supplies, printed circuit boards, and the CRT screens—in essence, the part one can see and touch. The other is software, sometimes thought of as the intelligence for the computer. Software is the set of commands or instructions that directs the hardware to perform in a certain way. The hardware has to be told when to add or subtract, when to initiate communications, when to print a file, etc. In some systems this can add up to millions of software instructions. Software engineering is the process whereby people design and implement these instructions to be used by the hardware. (Unless the term *computer hardware* is specifically used in this paper, the term *hardware* connotes generic systems hardware.)

Software engineering, unlike traditional forms of engineering, has no foundation in physical laws. The source of the structure for software engineering is in standards and policies, and these standards and policies are defined by teams of experts. For example, the languages used to generate instructions (FORTRAN, COBOL, Ada) are embodied in the language definitions. How data are stored is defined by the database structure and format, again a set of standards. There are few laws of nature, per se, for software to rely on; this is the strength and the weakness of software. It is very flexible and can be easily redefined. The developer can violate many of the laws (standards and policies) whenever it is convenient. This has made it difficult to develop a set of standards and statistics that allows people to measure the effectiveness or quality of software until after it is built.

Because software has foundation only in mathematics and logic and not in physical laws, the ability of a software engineer to introduce uncertainty into a software system is greater than in any other field. McDermid [28] argues that "software development is, or should be, an engineering discipline," because "one of the characteristics of established engineering disciplines is that they embody a structured, methodical approach to developing and maintaining artifacts." The individuals involved have control over the standards and policies that govern software; they can, and often do, interpret these laws differently. *Thus, effective control of these uncertainties introduced during the software development cycle should be through very stringent management.* This has not been the case; to date there has not been a well-defined process for software development.

Since software development, in the majority of cases, is an ad hoc process, it is not surprising that the risk identification and management process has been by and large ad hoc also. That process, however, can be made systematic and structured even if the software development process is not. Computers have become such an integral part of most, if not all, engineering systems that their contribution to any overall system failure must be addressed explicitly and, to the extent possible, quantitatively. Furthermore, the advances in hardware technology and reliability and the seemingly unlimited capabilities of computers render the reliability of

Manuscript received April 24, 1992; revised August 5, 1992, October 17, 1992, and December 1, 1992. This work was supported by the Department of Defense.

C. Chittister is with the Software Risk Management Program at the Software Engineering Institute, Carnegie Mellon University, Pittsburgh, PA 15213.

Y. Y. Haimes is with the Center for Risk Management of Engineering Systems at the University of Virginia, Charlottesville, VA 22903.

IEEE Log Number 9209624.

most systems to be more heavily dependent on the integrity of the software used. Thus, software failure must be scrutinized with respect to its contribution to overall system failure and with the same diligence and tenacity that have been devoted to hardware failure. *Failure* is defined as the nonperformance of some action that is due or expected, often used interchangeably with the term *malfunction* [22].

> *Risk is defined here as a measure of the probability and severity of an adverse event.*

For example, the risk of cost overrun is a measure of the probability and amount of cost overrun.

> *Software technical risk is defined as a measure of the probability and severity of an adverse event inherent in the development of software that does not meet its intended functions and performance requirements.*

The Defense Systems Management College [8, p. 3] has a similar definition of risk as "the probability and consequence of not achieving some defined program goal—such as cost, schedule, or technical performance."

Risk assessment and management is defined as a disciplined and systematic process of identifying, measuring, evaluating, confronting, and resolving (controlling) risk. While risk cannot be eliminated by practicing risk management, its consequences can be evaluated, prepared for, and ultimately managed. In a landmark evaluation of policies and procedures for technical risk assessment in the Department of Defense, the General Accounting Office made the following quintessential statement, which is as appropriate for software development as it is for new weapon systems [11]:

> Technical risks are inherent in the development of new weapons systems whose performance requirements exceed the capabilities of current weapons systems. If not anticipated and managed early in the acquisition process, these risks can have profound effects on a program's cost and schedule and, ultimately, the effectiveness of the armed forces.

Because of the high degree of interaction between hardware and software in a system, it is difficult to determine whether the source of failure is computer hardware- or software-based. The following four risks are most common during the software development process—cost overruns, schedule slips, not meeting requirements (what is to be built), and not meeting technical specifications (how it will be built).

The assessment and management of risk associated with the development of software engineering has been gaining increasing attention in books and numerous articles, all of which give prominence to the tutorial volume on the subject by Barry Boehm [3]. Indeed, this book guides the reader through the fundamentals of software risk analysis. Humphrey [20] offers a comprehensive treatise on the software development process. Sage and Palmer [34] emphasize the "systems engineering and systems management for software productivity perspective." In a book, Charette [5] states: "This book provides us with the tools, but tools alone do not make a competent software engineering risk analyst. We need to understand the greater context in which software engineering resides, and the risks and limitations within software engineering itself." In a second book, Charette [6] "focuses on a broader perspective than just the mechanics of risk analysis and management, by examining the totality of the software engineering enterprise, with all of its associated risk." Freeman [10], although he does not address software risk explicitly, does provide a valuable framework with which one can better understand the intricacy of software development.

The sources of risk associated with software development are many and varied. Indeed, at each stage of the software life cycle (design, development, testing, installation, integration into a larger system, and its ultimate use), one can identify numerous sources of risk. In this paper, however, the focus is on assessing the likelihood and consequences of failure in development, i.e., the risk associated with the entire process of software development.

Studying the risk associated with software development requires the analyst to consider all sources of system failure introduced during the development process—hardware, software, organizational, and human [15]. Building on significant past contributions to the field, Boehm [4] presents in a more recent paper basic principles and highlights acceptable practices in software risk management. He notes very perceptively that:

> one pattern that emerged very strongly was that the successful project managers were good *risk managers.* Although they generally didn't use such terms as "risk identification," "risk assessment," "risk-management planning," or "risk monitoring," they were using a general concept of risk exposure (potential loss times the probability of loss) to guide their priorities and actions. And their projects tended to avoid pitfalls and produce good products.

Software risk management also requires applying the best available knowledge, which leads one to ask: What is the state of knowledge of the identification, quantification, analysis, evaluation, and management of the risk associated with software engineering development? To do justice to answering this question, it is instructive to first address two surrogate questions: What distinguishes software development from hardware development? And, what is the state of the art of risk assessment and management of hardware systems as opposed to software systems? A by-product of these two questions is yet a third question, one that is more fundamental and central to the theme of this paper: To what extent can modeling, assessment, and risk management technologies that are derived from the development of hardware engineering systems be transferred to the software field? The following sections provide some philosophical and definitional perspectives in addressing these questions. Clearly, there is a need to develop a set of methodologies and processes to facilitate the assessment and management of risk in software development. In this context, Chittister, Kirkpatrick, and VanScoy [7] state "Until we use a disciplined and systematic method to identify and confront technical risk, we will never be able to control the quality, cost, or schedule of our software products." This is the driving force behind this paper. Indeed, in the Introduction and Overview to Part I of the *Software Engineer's Reference Book,* edited by John A. McDermid [28], McDermid and

Denvir identify myriad problems associated with software development. Among these are "... that software is late, over budget, incorrect, incomplete and does not meet requirements." They argue that among the following three perceived solutions: 1) to improve quality assurance procedures, 2) to organize the development process more systematically, and 3) to adopt a more "scientific" or "mathematical" approach, the third "solution," a scientific or mathematical approach, is the one of interest and constitutes the bulk of Part I of their reference book. Although this paper pays special attention to "solutions" 1 and 2, it also concentrates on the third, more scientific or mathematical, approach.

II. Background

The road to building a software system is full of surprises. The dilemma is that the people involved appear to spend a great deal of time convincing each other that the most recent surprise was the final surprise, and they are continually proven wrong. Software often goes through numerous changes, upgrades, fixes, recompiles, and system builds, etc., to address problems; nevertheless, new problems invariably arise. These changes take place because requirements and the system change continuously, people make mistakes, hardware manufacturers make changes to the system in response to marketing information, engineers introduce improvements and make commercial-off-the-shelf (COTS) software changes. And because often there is a break in communication, all these changes necessarily introduce uncertainties into the software development process and into the road to building software. Where does this uncertainty come from? Why can't software problems be identified and confronted before they become a crisis?

The ability to predict software problems beforehand has three major components:

1) Identifying and anticipating problems before they happen.
2) Determining the magnitude of potential or existing problems or risks.
3) Communicating the problems or risks to the appropriate people (people who cause, fix, are affected by, or responsible for the problems).

The difficulty with identifying risks is that they come from many different areas. For example, there is a likelihood that the COTS software vendor will not deliver the product on time, that the software developers do not use the COTS product correctly, that the COTS product does not meet performance requirements, that the requirements change and the COTS product does not meet the new functionality required, or that another vendor changes its COTS product and the two products are no longer compatible. In these examples, neither the events nor their consequences, i.e., risks, were necessarily apparent at the time the decision was made to use the COTS product. This is another difficulty with risk identification—both the probability that an event will happen and its impacts if it happens are uncertain and difficult to measure or quantify. Impacts can depend on the perspective and have different magnitudes, e.g., impact on cost, schedule, performance, quality.

There is a similar situation that can occur with computer hardware. In this case, the original equipment manufacturer (OEM) provides hardware to the system developer. This hardware is then incorporated into the system developer's product. If the OEM does not supply the equipment in the scheduled time or if the equipment does not perform as specified, the result from a systems perspective is the same as the failure of a COTS product. For example, system developers have relied on chip manufacturers to deliver computer chips, and when these chips are either late or fail to perform, the system does not work. The hardware solution is to use second-source components, and this may be the ultimate solution for software. This is not yet practical for software because of the lack of standard components or agreement in the community as to what constitutes standard components for software.

Managers and technical leaders have been struggling to identify the risks associated with system design and development since the first system was built. Significant progress in engineering hardware has been made in the ability to build models and simulations, gather data, and do statistical analyses. In the software field, however, these approaches to risk identification and analysis have been met with mixed results. Indeed, the premise here is that until a disciplined and systematic way is developed and used to identify and confront software technical risk, it will be impossible to control the quality, cost, and schedule of software products. This does not imply that individuals do not employ risk management. What it does say, however, is that the community does not employ it the same way each time. Generally, project managers and technical leaders use their own experience to build a set of methods and techniques that fit their purposes on a particular project. The problem with this practice is that when they venture into a new area (such as software), their experience may not be broad enough to handle the risks in these areas. This situation is further exacerbated by the fact that software is a new area for most project managers, and software is assuming a larger role in meeting system requirements. In fact, in many new systems, software is the technology used to integrate the hardware into an overall system. It could also be argued that program managers with software systems expertise have difficulty in identifying and managing software risk because of the lack of a systemic approach to software risk management.

III. Software as a Component of a Larger System

The complexity of the goods and services delivered during the last two decades or so has led to extensive subspecializations in many fields, most notably in engineering. Although this subspecialization stems from our need for detailed expertise in narrow fields, it has brought with it a parochial and often limited vision of broader, overall system perspectives. Indeed, few engineers have either the opportunity or the expertise to appreciate their own contribution (within a project) to the entire system. One of the consequences of this prevailing trend has been an unprecedented growth in the number of professional societies and an even larger number of committees within each society. For example, the Institute of Electrical

and Electronics Engineers (IEEE), the largest professional organization in the world, has 37 member societies; the Systems, Man, and Cybernetics Society of the IEEE has 22 committees; the IEEE Computer Society has 32 technical committees and task forces. This proliferation of professional societies is a manifestation of the need for in-depth knowledge and expertise in specific fields to achieve technological competitiveness. It is also a warning sign, however, against a growing tendency among these professional organizations to believe that their field or discipline is so unique that it merits introducing its own jargon, leading in many cases to attempts to reinvent the wheel. The inescapable question, therefore, is whether the risk of software development is yet another new discipline. As software assumes more and more functionality in the system, however, the software engineer is having greater influence on the system.

To understand what software development risk is, and to contribute to its assessment and management through the transfer of knowledge from the hardware engineering field to the software engineering field, one must 1) recognize the salient features and differences between the development processes of hardware engineering and software engineering; 2) understand the role of software engineering within the entire system; 3) appreciate, in the context of design and development, the uniqueness of software failure as juxtaposed against hardware failure, recognizing the importance of all four sources of system failure: hardware, software, organizational, and human; and 4) be familiar with the process of risk assessment and management from a total systems viewpoint.

A. Hardware Versus Software: Similarities and Differences

If one were to compare the architectural design of a hardware product with the development of a software product, one would be struck by the following distinguishing characteristic: There would be, in general, a finite and unambiguous number of fundamentally different paths or design options for hardware development to meet a given set of design specifications. Indeed, the extensive use of fault-tree analysis builds on this premise of finiteness. This is not so in the case of the architectural design of software; the number of significantly distinguishable paths or design options of software, for any given specifications, is significantly larger, more ambiguous, and broader. This inherently large number of degrees of freedom in design defies attempts to rely on historical statistics in predicting potential defects, faults, and errors in the development of software. (These terms are defined in the Appendix.) Because the situation is markedly different in the design of hardware products from software development (separate from and in addition to material defects), hardware design is substantially better able to prevent, detect, and correct defects, faults, and errors during development. McDermid [28] has this to say on the nature of software:

> Software has no physical existence and, in general, few reliable software metrics have been used. Software may have a much greater complexity than hardware and often includes highly structured data as well as logic. It is deceptively easy to introduce changes into software, the effects of which can propagate explosively.

As much as engineering design is perceived by some to be more of an art than a science, detailed and formal protocols and procedures that have been developed over the years continue to guide the work of engineering designers through a systematic and well-organized process. These existing protocols and procedures adopted for engineering design also serve as a unifying mechanism whereby engineers from different schools, backgrounds, and disciplines develop final products with great commonality in specifications, quality, and safety requirements. These procedures also constitute the building blocks upon which an organizational and managerial framework is built to oversee such engineering design and development. This systematized process also ensures the quality of the designed product and provides an environment conducive to total quality control.

In general, the design and development of software do not follow a well-established set of protocols and commonly accepted procedures. Indeed, in most cases, each software development is envisioned as a unique and distinct product. This lack of a well-developed and acceptable protocol has major implications on several dimensions for the assessment of software development risk. A list of such deficiencies includes the *lack* of

- uniformity in a work plan for software design and development
- uniformity in estimating costs and schedules
- systemic detection and diagnosis of faults and errors in a timely manner during the various phases of the development process
- development of fault-tolerant software design (and fault masking)
- total quality control
- overall risk assessment and management
- a common set of review methods
- standards against which to measure completeness or quality in the system
- understanding of the effect of changes on system performance

This list does not suggest that if software were done as hardware, then much of the risk would be reduced (or at least controlled). Rather, the list reinforces the need for the establishment of well-defined and effective procedures and protocols for software development and ultimately for software risk management.

B. The Role of Software Engineering Within the Entire System

As previously discussed, software is assuming a significant role in meeting system requirements. As this role grows, the impact of software on system risk grows. To be effective and meaningful, risk management must be an integral part of overall system management. This is particularly important in the management of technological systems, especially software-intensive systems, where the failure of a system can be caused by the failure of hardware, software, the organization, or its people.

The term *management* may vary in meaning according to the discipline involved and the context. As stated earlier, risk is defined as a measure of the probability and severity of adverse effects [26]. *Risk management* is commonly distinguished from *risk assessment*, even though some may use the term risk management to connote the entire process of risk assessment and management.

In the risk assessment component, the following three basic questions must be posed and answered at each stage of the software development process: 1) What can go wrong? 2) What is the likelihood that it will go wrong? 3) What would the consequences be? [23]. Only then can the next step commence: namely, answering the question, "What can be done?" This would include developing various design options; evaluating the various trade-offs among them; and ultimately selecting one or more acceptable options in terms of cost, reliability, performance, total quality, and safety. To answer the first three questions in the risk assessment process, however, one may benefit from following the paths of the four major sources of failure of systems in general, as well as in software development:

1) hardware failure
2) software failure (includes software used in the development of software)
3) organizational failure
4) human failure.

C. System Failure

Needless to say, the failure of a system caused by any of the above sources should ultimately be viewed with the same critical importance. This is because the outcome of a system failure induces the same impact regardless of the sources of failure. Organizational and human failures are endemic to software development failure and, thus, special attention and concern must be paid to them in the risk assessment process. Examples of such failures include overlooking and ignoring defects; being tardy in correcting defects; missing signals or valuable data due to inadequate testing; cutting corners to increase productivity (misguided trade-offs between productivity and safety); covering up mistakes due to competitive pressure; lacking incentives to find problems; the "killing the messenger of bad news" syndrome instead of "rewarding the messenger," screening information, followed by denial; and accepting the most favorable hypotheses. In her study of the failure of offshore oil drilling platforms, Pate-Cornell [30] identified many of these pitfalls. Thus, in addition to software experts, social and behavioral scientists and management and organizational experts should play a central role in the risk assessment of software development. For example, the fact that software managers are often too optimistic may have its roots in 1) human behavior (professionals are often too confident of their abilities and knowledge); 2) the elusive nature of the software design process itself; 3) the lack of protocol/process (e.g., designers do not follow a unified step-by-step process with well-established protocols and do not adhere to Gantt charts); and 4) the relatively lower level of performance accepted for software development as compared with hardware development. (Many manufacturing companies, e.g., Motorola [35], are aspiring to a 6-σ quality control, implying in some sense less than four failures in one million, but software design is not subjected, as yet, to such high standards of performance.) In many cases, of course, only a zero number of defects makes sense for software.

D. Total Risk Management

Total risk management (TRM) can be defined as a systemic, statistically based, and holistic process that builds on formal risk assessment and management. It answers the previously introduced two sets of triplet questions for risk assessment and risk management, and it addresses the set of four sources of failures within a hierarchical-multiobjective framework [15]. Needless to say, total management and total risk management are synonymous with a holistic philosophy and a systemic approach to management. More is said about TRM in software development in subsequent sections.

In our quest to develop a methodological framework for risk assessment and management of software development, we borrow from the philosophy and concepts developed in systems engineering, engineering design, and quality control to supplement and complement the state of the art of risk assessment and management. Building on common characteristics, we acknowledge the fundamental differences between the design and manufacture of hardware and the design and development of software. The engineering professional wants software engineering to adhere to its concept of system design and implementations. This means that software engineering should be a process whereby components are organized in a systematic and structured way to produce a product. The product design is documented and then goes to manufacturing. A central function of engineering is to produce a set of documentation on the basis of which others can manufacture a product. Software engineering follows a similar function, but has some distinct differences. There is no manufacturing stage per se; it is built into the original design process. Also, software does not always have a prior stage where research and development (R&D) is done. Software design is more analogous to integrated hardware R&D, system design, and manufacturing. For this reason, concepts of quality control, testing, product verification, and system integration have to be both broader and more encompassing for software than for hardware. In software there are usually no standard sets of components or building blocks to take advantage of in the development process. Advancement is being made in this area, however, with the introduction of products such as Ada, computer-aided software engineering (CASE) tools, and software engineering environments.

In the design of hardware systems, a designer who is faced with a specific problem generally follows a well-established protocol/process for that design. For each stage within that protocol (e.g., economic feasibility, technical feasibility, prototype), the designer selects an appropriate methodological approach from personal experience. All this is done while attending to the specific problem characteristics and environmental and other external forces.

The creation of a systemic protocol/process for software design and development is similar to engineering design. Such a fundamental step would then chart the course for the conduct of risk assessment for software development. The advantages that result from such a protocol/process lie in the statistical area. One often encounters the dilemma of a lack of statistical data on software failure that can be related to and transferred from one case to another. There is evidence that this problem is beginning to be addressed in some quarters, e.g., NASA at Goddard is currently collecting and analyzing project data. This approach, which is aimed at adding value to existing statistical data such as the effort to define software metrics, could also shed light on and benefit from the further development of measures that characterize software (e.g., complexity, number of lines of codes).

It is evident from the above discussion that although experiences and knowledge from the hardware manufacturing field can be beneficially transferred into the development of engineering software, the intrinsic differences between these two products *mandate the introduction of a modified, if not different, paradigm for the prevention, detection, and correction of faults and errors associated with the development of software.* This is the challenge facing the software engineering community.

Furthermore, quality management of hardware manufacturing differs, in a fundamental way, from that of the development of software engineering. This is because the process of prevention, detection, and the ultimate correction of faults and errors during the development of software is driven by myriad human factors and production procedures, an organizational culture, and architectural design characteristics that often have no parallel in the production of hardware. In spite of these differences, a significant amount of experience can be transferred (with respect to quality management) from the manufacturing of hardware to the development of software. This optimism is based primarily on the centrality of human factors in such a transfer as is discussed in subsequent sections. Finally, identifying and understanding the origin and implications of the differences between the two kinds of products can avoid fallacious transformation of product-specific knowledge from one case to another.

IV. Human Factors and Production Procedures

Quality control, according to Japanese axiom, starts and ends with education and training. Indeed, one may view all four sources of system failure—hardware, software, human, and organizational—as being directly affected by education and training. Continuous process improvement, adherence to total quality control, developing and sustaining an organizational culture that promotes product quality through continuous process improvement—none of these aspirations can be realized unless and until the work force is appropriately educated and trained. These principles, which are becoming more and more ingrained in today's engineering design and manufacturing, should not be overlooked in software design and development. In fact, education and training seem to have an even more dominant role in assuring the quality of software than in hardware, given the intricacy and centrality of human involvement in software design and development and the lack of physical manifestations of the invisibility of software. In other words, the source of failure due to human and/or organizational errors is more prevalent in software than in hardware design and development. This necessarily leads to the importance of developing a curriculum for software development that is grounded not only in computer science and engineering, but that also draws from the social, behavioral, and management sciences.

The vast technical literature on hardware manufacturing—manuals, handbooks, textbooks, and journals—constitutes a solid foundation upon which a designer or a manufacturing engineer can rely. The lack of current comparable specific technical guidance for architectural design, coding, and development of software engineering prevents the software professional from relying on the documented experience of others. This lack is particularly critical because *software development is an intellectual, labor-intensive activity.* Consequently, the absence of universally accepted and adopted production procedures for software development significantly raises to a higher level the relative contribution of human factors to software faults and errors. The relative unavailability of well-tested development procedures also provides a welcome personal challenge to the involved professionals in software development. And, to compensate for this absence, software professionals build more on their creative and imaginative genius than is the case in hardware manufacturing. In a book titled *Software Engineering: A Holistic View,* Blum [2] argues that

> as a problem-solving, modeling discipline, software engineering is a human activity that is biased by previous experience. That is, unlike the modeling of physical phenomena, there are few confirmable answers bounded by invariant constraints. Lacking an external reality, we must conduct the process in ways that seem reasonable and work. Consequently, it is as important to understand why we do what we do as it is to learn how to do it.

This shift from well-established procedures in hardware manufacturing to a more ad hoc artistic human involvement in software development increases the likelihood of the introduction of more faults and errors in software development and presents new challenges in the quest for the prevention, detection, and correction of these faults and errors. Thus, it is imperative to develop, nourish, and sustain qualified personnel—people with high self-esteem, who identify themselves with the organization and its success, and who have a serious commitment to their jobs and to their fellow workers.

Apathy and indifference among workers are sources of trouble and failure. Masaaki Imai [21] shares with his readers a very popular term in Japanese total quality control activities—*warusa-kagen*—which refers to things that are not yet problems, but are still not quite right. Left unattended, they may develop into serious problems. He states:

> *Warusa-kagen* is often the starting point of improvement activities. In the workplace, it is usually the worker who

> notices "*warusa-kagen*," and, hence, the worker becomes the first echelon of maintenance and improvement.

Encouraging and promoting *warusa-kagen* at all stages and levels of software design and development is a recipe for ultimate risk management.

Developing a sustainable system of continuing education and training, not only for the technical staff but also for management, is an integral part of human resource development. Such an effort should not be thought of as a discrete activity that takes place once or twice a year. The multifarious nature and complexities of software engineering design, development, and management have been addressed and "managed" over the years by computer scientists and engineers, but not by experts in business administration. Computer science and computer engineering programs at various universities rarely associate with the schools of business and management. Yet many of the basic ills and problems in software engineering are the result of ineffective management.

Most industrial nations that believe in the importance of technological leadership and the role of manufacturing in economic competitiveness have adopted strict *and* strong quality control management policies. Have software designers and developers benefited from this knowledge? Deming's 14 points [9] have become a milestone in the culture that many organizations have embraced. Continuous improvement that squarely focuses on the development of human resources at all levels of management and personnel has become the cornerstone of the organizational culture for many successful companies.

In his book *KAIZEN* (which means improvement), Masaaki Imai [21] articulates the ingredients for successful management. He states:

> Improvement as a part of a successful *KAIZEN* strategy goes beyond the dictionary definition of the word. Improvement is a mind-set inextricably linked to maintaining and improving standards.

On the subject of quality, Imai states:

> There is very little agreement on what constitutes quality. In its broadest sense, quality is anything that can be improved. When speaking of 'quality' one tends to think first in terms of product quality. When discussed in the context of *KAIZEN* strategy, nothing could be further off the mark. The foremost concern here is with the *quality of people.*
>
> The three building blocks of a business are hardware, software, and "humanware." Only after humanware is squarely in place should the hardware and software aspects of a business be considered. Building quality into people means helping them become *KAIZEN* conscious.

Although the management concepts advanced in *KAIZEN* address issues in manufacturing, they are equally germane and relevant to software engineering design, development, quality control, and risk management.

While one might argue that procedures for the promotion and nourishment of an appropriate culture of an organization that manufactures hardware products can be duplicated for organizations that develop software engineering, the prevalence of basic differences is evident. First and foremost is the relatively short period in which software development has moved from a small-scale venture to large-scale industry with the involvement of hundreds or even thousands of professionals in the development of software within a single company. The relatively short time in which this jump in the magnitude of the size of software development firms has taken place has not allowed, in most cases (Hewlett-Packard, an established hardware manufacturer who has applied good engineering principles to software development, is one of the exceptions to this trend [12]), these firms to develop the type of organizational culture advocated by Deming and Imai.

V. A Decision Making Hierarchy for a Risk Management Framework

Students of software engineering can easily develop a long list of problems, issues, dilemmas, conflicts, and research needs associated with almost any software engineering development. Such lists are also abundant in numerous publications and technical reports. In our quest to develop an analytical framework for risk management of software engineering, however, we will focus on the *sources and causes* of these problems, attempt to group them into a meaningful, yet manageable, number of categories, and then develop a comprehensive framework for dealing with the causes rather than the symptoms. To streamline the discussion and add order to it, a hierarchical structure will be adopted. The architect Rafael Vinoly, whose new International Forum in downtown Tokyo is one of the decade's most prestigious design commissions, said of his design: "The thing that interested me about the Forum from the beginning is that you cannot box this building into a narrow category. It's a composite of different things that together make a unique structure" [25]. Indeed, it is impossible to do justice to a comprehensive framework for the risk assessment and management of software development by boxing it into one planar structure (model). By allowing cross-representations and overlapping models of the various facets and dimensions of the process, hierarchical holographic modeling (HHM) [14], [18], which is discussed subsequently, alleviates some of the limitations of a single schema or a single vision of the complex system.

In the abstract a mathematical model may be viewed as a one-sided limited image of the real system that it portrays. To clarify and document not only the multiple components, objectives, and constraints of a system but also its welter of functional, temporal, and other aspects, is quite impossible with single-model analysis and interpretation. Given this assumption and the notion that ever-present integrated models cannot adequately cover a system's *aspects* per se, the concept of HHM constitutes a comprehensive theoretical framework for systems modeling.

Fundamentally, hierarchical holographic modeling is grounded on the premise that large-scale and complex systems, such as software development, should be studied and modeled by more than a single way, vision, or schema. And, because such complexities cannot be adequately modeled or represented through a planar or a single vision, overlapping among these

visions is not only unavoidable, but can be helpful in a holistic appreciation of the interconnectedness among the various components, aspects, objectives, and decision makers associated with such systems.

Central to the mathematical and systems basis of holographic modeling (from the perspective of theoretical constructs) is overlapping among various holographic models with respect to the objective functions, constraints, decision variables, and input-output relationships of the basic system. In this context holographic modeling may be viewed as the generalization of hierarchical overlapping coordination (HOC) (see [13], [17], [27], [18]). The methodology of HOC provides a mathematical framework for representation of such systems, permitting alternative decompositions of a single system and showing how the mutual functioning of the resultant plurimodels may be coordinated. In the heraldically overlapping coordination, a system's single model is decomposed into several decompositions—in response to the various aspects of the system—where these decompositions are coordinated to yield an improved solution.

Several experts in software engineering have recognized the limitations of a single planar model for the management of large-scale and complex systems and the advantages of a multivisionary model in the context of hierarchical holographic modeling, even though the term HHM is not cited as such. Most notably among these works are papers by Yeh *et al.* [38] and by Leite and Freeman [24]. The article by Yeh *et al.* is quite explicit in its rejection of a single-model, step-by-step approach in managing complex software engineering development. The authors make a significant contribution to the understanding and modeling of complex software systems. They state:

> In most current practice, decisions are based on a one-dimensional view prescribed by waterfall-like models. This view consists of a single explicit perspective on a set of activities and their interdependencies and schedule—which form an *activity* structure. In the waterfall model, the activities are sequentially scheduled into phases, requirements analysis, design, codes, tests, and so on. Other models suggest adding a second perspective, a *communication* structure . . . still other models like the spiral model of software development and enhancement and models based on process-maturity levels suggest including process design and monitoring.

These very arguments detailing the limited capabilities of any single model to capture the multifarious aspects of a complex system were at the heart of the factors that compelled the development of the hierarchical holographic modeling framework. Yeh *et al.* describe their multimodel approach: "Our process-management model, called Cosmos, combines the best of these models, incorporating all three perspectives: activity, communication, and infrastructure. Cosmos is designed to manage a large software system from cradle to grave." The decision-making hierarchy for a risk management framework developed in this paper, which builds on the hierarchical holographic modeling construct, is congruent with the approach advanced by Yeh *et al.*

Management occupies the highest level of the hierarchical structure. The term *management*, however, may vary in meaning according to the discipline involved and the context. Total management, which is advocated here, means a philosophy that promotes and subscribes to

1) a holistic systems approach to problem solving
2) a resource allocation policy that considers, for example, human resource development as important as the architecture of the software design
3) *KAIZEN* and continuous improvement in their broadest sense
4) the development of human resources
5) appreciation of the role of and understanding of the interdependencies among
   a) the design and development of software
   b) the organizational infrastructure
   c) the quality and development of human resources
6) total risk management—a systemic approach for dealing with risk, uncertainty, and impact analysis

An overall hierarchical framework that incorporates the above six elements of total management is presented in subsequent discussion.

The analytical framework proposed here for total software risk management builds, to a large extent, on hierarchical holographic modeling, relying on the conceptual features of HHM and on its philosophical construct, not on its mathematical modeling innovation.

To avoid too much theoretical discussion (which is covered in Haimes [14] and Haimes *et al.* [18]), the stratagem presented here for risk identification revolves around three hierarchical levels. After each level and its associated subsystems are introduced, a more detailed and comprehensive discussion of the entire risk assessment structure is presented. In the context of hierarchical holographic modeling, several visions of the risk assessment and management framework for software development are presented (see Figs. 1–3). The three major decompositions (visions) are introduced first, followed by a discussion of their integration within an overall HHM structure.

A. Functional Decomposition

One of the visions or decompositions that constitutes the HHM for the risk assessment and management of software development is the functional perspective. From a functional perspective, the software development process may be decomposed into the following seven subsystems: requirement, product, process, people, management, environment, and the development system (see Fig. 1).

1) *Requirement:* The highest-level definition of what the product is supposed to do: what needs it must meet, how it should behave, and how the customer will use it. It corresponds to the production perspective.
2) *Product:* The output of the project that will be delivered to the customer. It includes the complete system: hardware, software, and documentation.

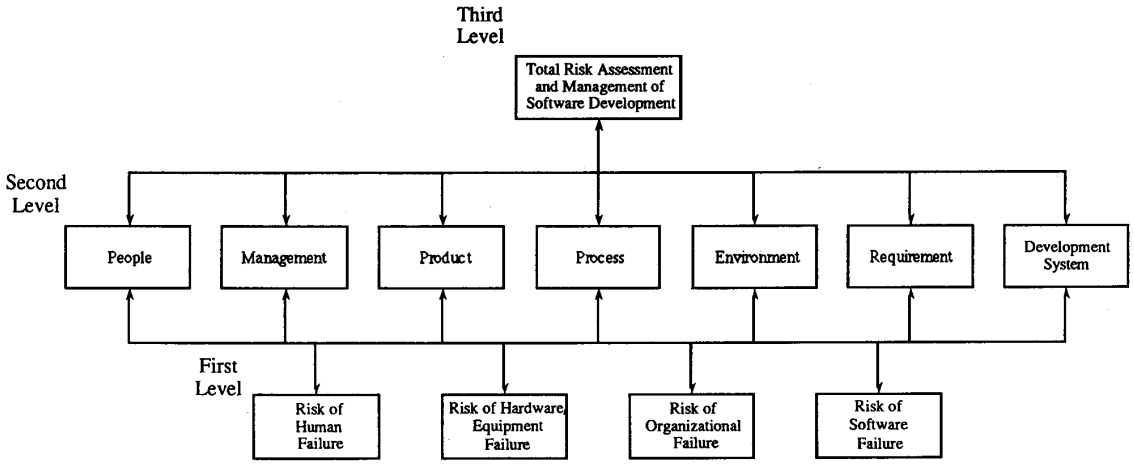

Fig. 1. Total risk assessment and management of software engineering: Functional-based hierarchical holographic structure.

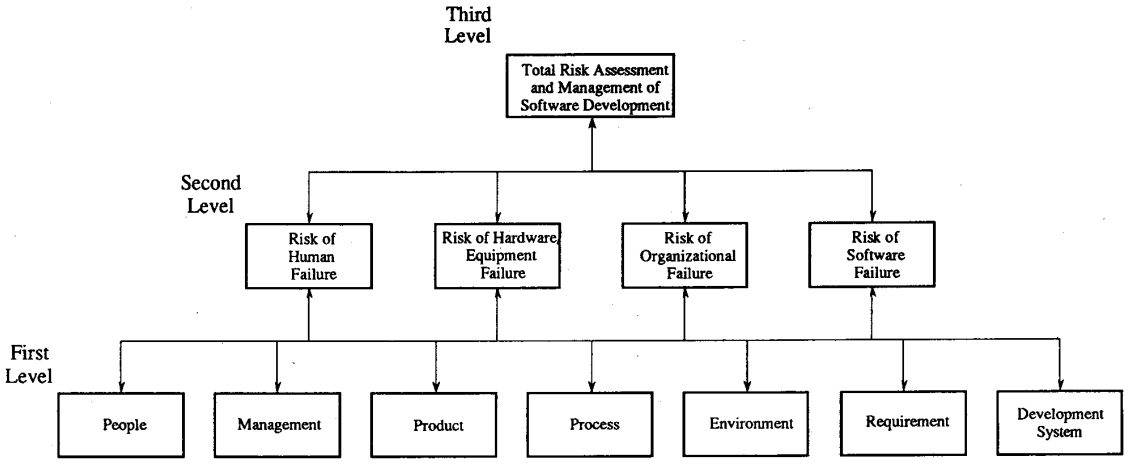

Fig. 2. Total risk assessment and management of software engineering: Source-based hierarchical holographic structure.

3) *Process:* The way by which the contractor proposes to satisfy the customer's requirement. The process is the sequence of steps—their inputs, outputs, actions, validation criteria, and monitoring activities—that leads from the initial requirement to the final delivered product. It includes such phases as requirements analysis, product definition, product creation, testing, and delivery. It includes both general management processes, such as costing, schedule tracking, and personnel assignment, and project-specific processes, such as feasibility studies, design reviews, and regression testing.
4) *People:* All those who will be associated with the technical work on the project and all the support staff. It also includes the technical advisers, overseers, and experts, whether in the chain of command or matrixed.
5) *Management:* The line managers at every level who have authority over the project, including those responsible for budget, schedule, personnel, facilities, and customer relations.
6) *Environment:* The "externals" of the project: the factors that are outside the control of the project but can still have major effects on its success or be sources of substantial risk.
7) *Development system:* The methods, tools, and supporting equipment that will be used in the product development. This includes, for instance, CASE tools, simulators, design methodologies, compilers, and host computer systems.

B. Source-Based Decomposition

Another vision of the HHM can be obtained through the four sources of system failure discussed earlier (see Fig. 2):

1) hardware failure
2) software failure (software used in the development of software)
3) organizational failure
4) human failure

These four sources of failure are not necessarily independent. Just as the distinction between software and hardware is not always straightforward, neither is the separation between

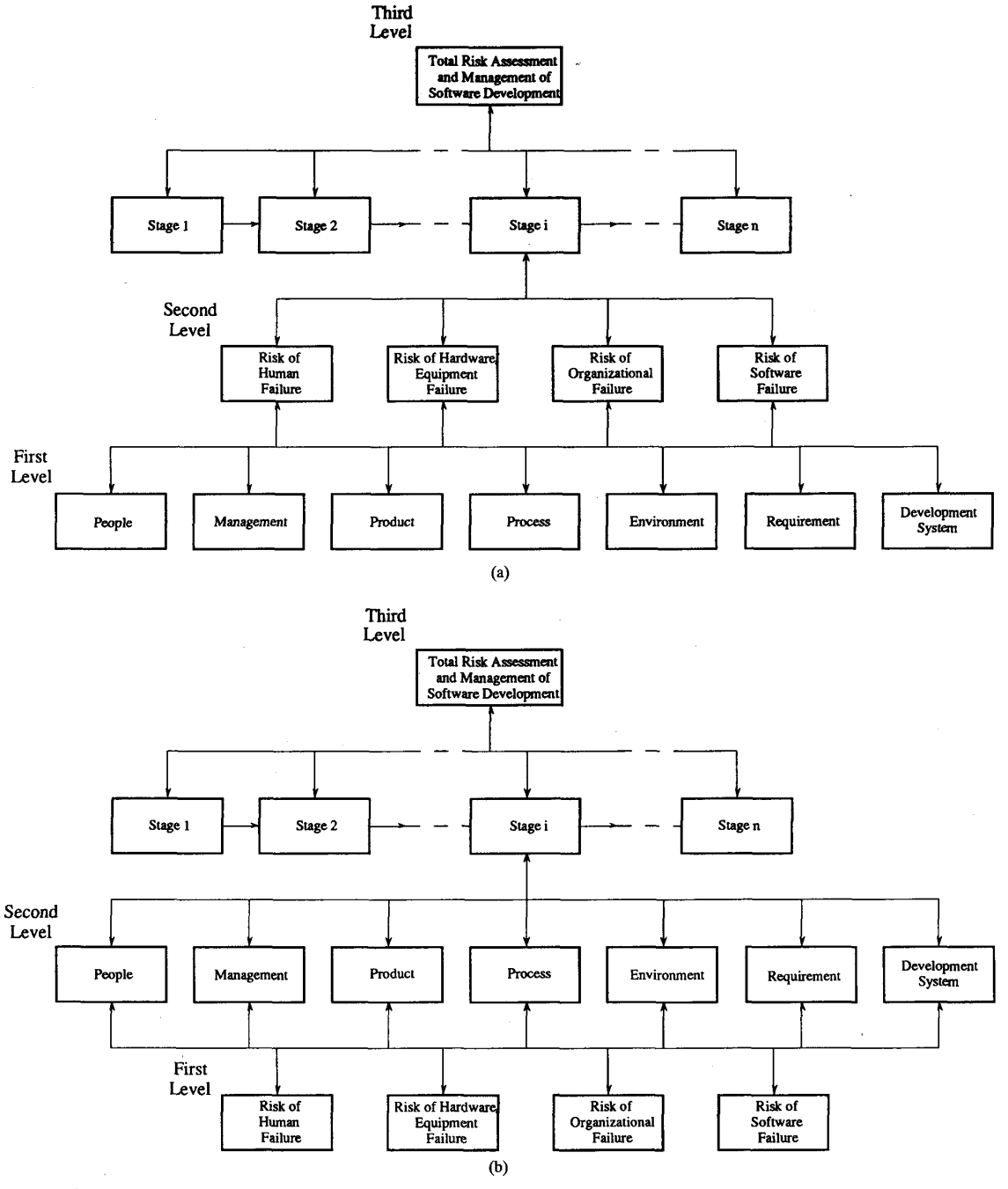

Fig. 3. (a) Total risk assessment and management of software engineering: Temporal-based hierarchical holographic structure. (b) Total risk assessment and management of software engineering: Temporal-based hierarchical holographic structure.

human and organizational failure. Nevertheless, these four categories of sources of failure provide a meaningful foundation upon which to build the decision-making hierarchy for the proposed framework. Note that software development is an intellectual, labor-intensive activity that must be streamlined through a well-managed organizational infrastructure and nurtured by an organizational culture and vision that are conducive to and driven by a *warusa-kagen* philosophy.

C. Temporal Decomposition

The third vision of the HHM relates to the evolution of software development over time. Each of the various overlap-

ping and iterative stages of software development, although often not sharply distinguishable, constitutes a subsystem in the temporal decomposition (see Fig. 3). For the purpose of this paper, the following temporal stages are identified [20]:

1) system requirements
2) software requirements
3) analysis
4) program design
5) coding
6) testing
7) operations

Each stage (subsystem) in the temporal decomposition can be viewed as one frame in a fixed time (e.g., testing) during the software development process. It is at this fixed-time frame that risks associated with the functional decomposition (e.g., requirement) and with the source-based decomposition (e.g., organizational failure) are identified and articulated. As another example, consider the following four risks that are common during each stage of software development: cost overrun, time delay, not meeting requirements, and not meeting technical quality specifications. The temporal domain has significance far beyond the schedule of the project; it articulates the changing and evolution of risks over time.

VI. Hierarchical Holographic Modeling for Risk Assessment

Each of the three hierarchical holographic (HH) submodels developed here contributes to the identification of risk associated with software development. The overlappings among these HH submodels mimic the fuzziness that characterizes the real world of software development with respect to the inability to make a clear distinction among the various causes of failures. Fig. 1, which is an inverted image of Fig. 2, presents an entirely different perspective in answering the set of triple questions: What can go wrong? What is the likelihood that it will go wrong? And what would the consequences be? Since a central objective of risk assessment is to identify, to the extent possible, everything that can go wrong, then a hierarchical holographic modeling structure is superior to a planar single model in this respect. Note, for example, that in Fig. 1, the four sources of risk (human, hardware, organizational, and software) are investigated for each subsystem of the functional decomposition (people, management, product, process, environment, requirement, and development systems). On the other hand, Fig. 2 depicts a different perspective; namely, the seven functional decompositions are investigated for each of the four sources of risk of failure. Fig. 3 incorporates the temporal decomposition that captures the stagewise evolutionary process of software development, and thus the risk associated with each stage and for each subsystem of the functional and source-based decompositions. In particular, Figs. 3(a) and (b) extend Figs. 1 and 2 by incorporating the temporal domain into the HHM. A more elaborate discussion of the role of HHM in the risk assessment and management of software development will be presented in subsequent papers.

The HHM framework discussed in the previous section and depicted in Figs. 1–3 provides a systemic methodological framework for identifying the universe of "what can go wrong." The application of the risk ranking and filtering (RRF) method [16] facilitates the process of distinguishing and discriminating among the potential failures or adverse effects. In a nutshell, the genesis for developing the RRF methodology can be summarized as follows:

The U.S. Department of Defense recently identified over 17,400 sites at Army, Navy, Air Force, and Defense Logistics Agency installations that present a potential for environmental contamination [32]. Each year thousands of mechanical and electronic components of the NASA Space Shuttle are placed on the critical item lists (CILs), an effort to identify items that contribute significantly to program risk [29]. The common aspect in these situations is the need to prioritize a large number of entities with respect to their individual contribution to a risk, which is a potential contribution to adverse consequences involving a larger system, for example, the Space Program or the natural environment. A prioritization of critical components or contaminated sites makes possible the implementation of a risk-reduction policy, or risk management by remedial action.

The RRF is guided by the following observations about prioritization efforts:

1) Risk is a composition of a multitude of probable adverse effects in which the unprocessed data can be subjective and objective, qualitative and quantitative.
2) In agreement with the partitioned multiobjective risk method (PMRM) [1], risks of extreme, moderate, and lesser consequences are not commensurated—they can and should be differentiated.
3) Data acquisition should realize an economy of cost and information requirements: typically less information is needed about each of the top 1000 items than about each of the top 20.
4) Measurement theory is a formalism to guide the quantification of human and machine observations. Also, quantitative decision tools are approximate by nature, and an appreciation of the sensitivity of any method to its inputs is essential.

The RRF methodology is divided into two major phases—the telescoping filter and the prioritization phase. The telescoping filter reduces the number of components under consideration to the top 20 critical items by refining attributes and adjusting the threshold successively. Once the list has been reduced to the top 20, or any predetermined number, the prioritization phase is used to rank order the list in decreasing order of contribution to program risk.

In the RRF methodology, the objective of minimizing program risk is made explicit and quantifiable. The following five criteria that contribute to program risk may be viewed as a representative sample: 1) prior risk information, 2) moderate-event risk, 3) extreme-event risk, (d) fault tolerance, and (e) risk reduction potential (optional).

A measurable attribute is assigned to each of these five criteria of the program risk hierarchy. An assessment process

with each of the resulting criteria scales defines mappings from the criteria scales (e.g., number of incidents) to discrete scales of *severity* (very low = 1 to very high = 5).

For every critical item, there exist two equivalent risk fingerprints: 1) a bar graph of severity versus program risk criterion, and 2) a bar graph of severity *percentile* versus program risk criterion. A threshold, or decision rule, is specified on either one of the fingerprints and serves to reduce the size of the list—all items below the threshold do not pass the filter.

This process is applied repeatedly, in a telescoping fashion, to narrow the list by stages to the top 20. In successive stages, either the threshold is made more restrictive or more information is elicited to refine the representation; measurement efforts, and hence the filter, become less crude as the process advances stagewise.

Finally, when the last 20 items remain, a weighted sum of severity scores (adapted from the analytic hierarchy process), the risk fingerprints, and the criteria hierarchy provide a decision-support environment for the top-20 prioritization. It is recommended that weights attached to the five criteria be varied in an interactive procedure to explore the sensitivity of the final ordering to the weighting.

The flexibility of RRF is evident from the following. The telescoping filter can narrow the list to a size other than 20, subject to the prerogative of the decision makers; and reduction to *any specified number* of items can be fixed to occur after one or more stages of the process. Additionally, some subset of the five risk criteria can be used in early telescoping stages to restrict unnecessary data collection. Moreover, in the final phase, the weighted sum (developed through application of the analytic hierarchy process (AHP) [33]) alone can order the top 20, independent of the full decision support approach. The program risk hierarchy stays intact irrespective of the allowable modifications, thereby ensuring representation of the full contingent of program risk criteria throughout.

In every step of the RRF process, one works with clear and consistent definitions of the multiple criteria of risk and an intuitive, graphical representation of information and analysis. The RRF adopts two multiobjective and risk decision tools: the AHP and a risk assessment matrix originally developed for software risk [36]. In application to the Space Shuttle, the RRF has incorporated concepts from fault-tolerant design and extreme-event risk.

Applying the RRF method to the universe of "all" potential faults, errors, or failures identified in the software development process through the HHM framework provides a systemic filtering and ranking mechanism. The by-products are lists of software technical and nontechnical risk events with their associated priorities. The analyst at this stage may consider applying appropriate methodologies, such as decision trees [31], fault trees [37], fault tolerance [22], program evaluation and review technique (PERT), and critical path method (CPM) [19] to quantify further each of the n top risk events generated and prioritized by the RRF method.

VII. Conclusion

Software will continue to grow in size, complexity, and importance as it assumes more functionality in large, complex systems. If engineers and managers working in the community do not embrace a risk management ethic for software development, then software problems will continue to grow as well. Although practicing risk management does not guarantee fewer problems, it does provide a structure with which to make better decisions about the uncertainty and impact of future events. If risks can be measured, then contingency strategies can be provided; however, if risks are unknown, then surprise is likely when it is least convenient.

Although software engineering is different from other engineering disciplines, the management of risk in the developmental process is critical for all engineering disciplines. This paper has provided a framework for identifying and assessing risk in the software development process. The framework is grounded on the premise that software development is an intellectual, labor-intensive activity, thus making the human factor central to the assessment and management of risk. The framework builds on the concepts of hierarchical holographic modeling, total risk management, and *KAIZEN* (continuous improvement).

As systems become larger and more complex, the assessment and management of risk must be a team effort. The team has to consist of, among others, the system developers, support staff from the organization, and management. Risk management is not just the program manager's job and is not just a technical issue. Financial and quality risks are as important as software technical risk.

The more diversified the team, the more important it is to have a common and agreed-upon risk assessment and management process. The members of the team will have their own technical jargon and their own frames of reference. If each subgroup identifies and manages risks differently, there will be no common ground for communication or measurement. A systematic and structured process that is used by everyone will provide a foundation for discussion and for mitigation strategies. This process will also greatly reduce confusion caused by misunderstanding, which is itself a source of risk in large complex systems.

No one would argue that the best way to manage problems is to keep them from happening. A risk ethic that is embraced and practiced by an entire organization will significantly reduce the chaos created by unknown risks and crisis situations.

Appendix
Definition of Terms

The primary purpose of papers and articles is to communicate with the reader. This communication is more difficult in an article that transcends several disciplines, as this one does. To ensure that terms commonly used in one discipline denote the same thing to readers from other disciplines, we offer a set of definitions that may not be necessarily universal.

Error: The manifestation of a fault; a deviation from accuracy or correctness [22].

Failure: The nonperformance of some action that is due or expected; often used interchangeably with the term *malfunction* [22].

Fault: A physical defect, imperfection, or flaw that occurs within some hardware or software component [22].

Organizational Failure: The nonperformance of some expected action that can be attributed primarily to the mismanagement of an organization.

Reliability: The conditional probability that a system will perform correctly throughout the interval $[t_0, t]$, given that the system was performing correctly at time t_0 [22].

Risk: A measure of the probability and severity of adverse effects [26].

Risk Assessment: A process that attempts to answer the following set of triplet questions: What can go wrong? What is the likelihood that it will go wrong? What would the consequences be? [23].

Risk Management: Builds on the risk assessment process by seeking answers to three questions: What can be done? What options are available and what are their associated trade-offs in terms of all costs, benefits, and risks? What are the impacts of current management decisions on future options? [15].

Safety: A normative, political activity that judges the acceptability of risk; a thing is safe if its risks are judged to be acceptable [26].

Software Development Process: The process by which user needs are translated into a software product. The process involves translating user needs into software requirements, transforming the software requirements into design, implementing the design in code, testing the code, and sometimes installing and checking out the software for operational use (IEEE standards 610-12-1990).

Software Development Risk: A measure of the probability and severity of adverse effects (failing to meet expected performance) that can take place during the software development process, e.g., risk of cost overrun, risk of time delay in project completion schedule, risk of not meeting technical specifications. Note that the term *risk* connotes a multitude of risks.

Software Engineering: The application of a systematic, disciplined, quantifiable approach to the development, operation, and maintenance of software; that is, the application of engineering to software (IEEE standards 610-12-1990).

Software Engineering Process: The total set of software engineering activities needed to transform a user's requirements into software [20].

Software Technical Risk: A measure of the probability and severity of adverse effects inherent in the development of software that does not meet its intended functions and performance requirements.

Uncertainty: A lack of certitude ranging from a small falling short of definite knowledge to an almost complete lack of it or even any conviction, especially about an outcome or result (*Webster's Third New International Dictionary* 1986).

ACKNOWLEDGMENT

We would like to thank Bob Kirkpatrick, Jim Lambert, Dick Murphy, Vijay Tulsiani, and Julie Walker for their valuable comments and suggestions. We also appreciate the technical editing assistance provided by Sandra Bond. We are very grateful to the valuable and constructive comments and suggestions that we have received from the anonymous reviewers, the associate editor, and the editor of these *IEEE SMC Transactions.*

REFERENCES

[1] E. Asbeck and Y. Y. Haimes, "The partitioned multiobjective risk method," *Large Scale Syst.*, vol. 6, pp. 13–38, 1984.
[2] B. I. Blum, *Software Engineering: A Holistic View.* New York: Oxford Univ. Press, 1992.
[3] B. W. Boehm, *Tutorial: Software Risk Management.* Washington, DC: IEEE Computer Press, 1989.
[4] B. W. Boehm, "Software risk management: Principles and practices," *IEEE Software,* vol. 8, Jan. 1991.
[5] R. N. Charette, *Software Engineering Risk Analysis and Management.* New York: McGraw-Hill, 1989.
[6] R. N. Charette, *Applications Strategies for Risk Analysis.* New York: McGraw-Hill, 1990.
[7] C. Chittister, R. Kirkpatrick, and R. VanScoy, "Risk management in practice," *Amer. Programmer,* vol. 5, no. 7, Sept. 1992.
[8] Defense Systems Management College, *Risk Assessment Techniques: A Handbook for Program Management Personnel.* Ft. Belvoir, VA, 1983.
[9] W. E. Deming, *Out of the Crisis.* Cambridge, MA: MIT, Center for Advanced Engineering Study, 1990.
[10] P. Freeman, *Software Perspectives: The System Is the Message.* Reading, MA: Addison-Wesley, 1987.
[11] General Accounting Office, *Technical Risk Assessment.* Washington, DC, 1986.
[12] R. B. Grady and D. L. Caswell, *Software Metrics: Establishing a Company-Wide Program.* Englewood Cliffs, NJ: Prentice-Hall, 1987.
[13] Y. Y. Haimes, *Hierarchical Analyses of Water Resources.* New York: McGraw-Hill, 1976.
[14] ——, "Hierarchical holographic modeling," *IEEE Trans. Syst., Man, and Cybern.,* vol. SMC-11, pp. 606–17, Sept. 1981.
[15] Y. Y. Haimes, "Total risk management." Editorial in *Risk Analysis: An Int. J.* vol. 11, no. 2, pp. 169–171, 1991.
[16] Y. Y. Haimes, J. Lambert, D. Li, J. Pet-Edwards, V. Tulsiani, and D. Tynes, "Ranking of space shuttle FMEA/CIL items: The risk ranking and filtering (RRF) method," Tech. Rep. submitted to the Vitro Corp. and NASA headquarters, Center for Risk Management of Engineering Systems, Univ. Virginia, Charlottesville, VA, 1991.
[17] Y. Y. Haimes and D. Macko, "Hierarchical structures in water resources systems management," *IEEE Trans. Syst., Man, Cybern.,* vol. SMC-3, no. 4, 1973.
[18] Y. Y. Haimes, K. Tarvainen, T. Shima, and J. Thadathil, *Hierarchical Multiobjective Analysis of Large Scale Systems.* New York: Hemisphere Publishing, 1990.
[19] F. S. Hillier and G. J. Lieberman, *Introduction to Mathematical Programming.* New York: McGraw-Hill, 1990.
[20] W. S. Humphrey, *Managing the Software Process.* Reading, MA: Addison-Wesley, 1990.
[21] M. Imai, *KAIZEN: The Key to Japan's Competitive Success.* New York: McGraw-Hill, 1986.
[22] B. Johnson, *Design and Analysis of Fault-Tolerant Digital Systems.* Reading, MA: Addison-Wesley, 1989.
[23] S. Kaplan and B. J. Garrick, "On the quantitative definition of risk," *Risk Analysis,* vol. 1, no. 1, 1981.
[24] J. Leite and P. Freeman, "Requirements validation through viewpoint resolution," *IEEE Trans. Software Eng.,* vol. 17, Dec. 1991.
[25] P. Lemos, "Winning by design," *Vis a Vis,* vol. 6, no. 80, 1991.
[26] W. W. Lowrance, *Of Acceptable Risk.* Los Altos, CA: William Kaufmann, 1976.
[27] D. Macko and Y. Y. Haimes, "Overlapping coordination of hierarchical structures," *IEEE Trans. Syst., Man, Cybern.,* vol. SMC-8, Oct. 1978.
[28] J. A. McDermid, Ed. *Software Engineer's Reference Book.* Oxford, England: Butterworth Heinemann, 1991.
[29] National Space Transportation System (NSTS), "Instructions for preparation of critical item risk assessment (CIRA)," NSTS 22491, June 19, 1987.
[30] E. Pate-Cornell, "Organizational aspects of engineering system safety: The case of offshore platforms," *Science,* vol. 250, pp. 1210–1217, 1990.
[31] H. Raiffa, *Decision Analysis.* Reading, MA: Addison-Wesley, 1977.
[32] M. W. Read and J. M. Hushon, "Prioritizing DOD hazardous waste sites for cleanup," Office of the Deputy Assistant Secretary of Defense (Environment), N. Washington St., Alexandria, VA, and ERM—Program Management Company, Jones Branch Drive, McLean, Virginia, 1991.

[33] T. L. Saaty, *The Analytic Hierarchy Process.* New York: McGraw-Hill, 1980.
[34] A. P. Sage and J. D. Palmer, *Software Systems Engineering.* New York: Wiley, 1990.
[35] J. Wolak, "Motorola revisited," *Quality,* vol. 5, pp. 18–24, 1987.
[36] U.S. Air Force, "Software risk abatement," *Tutorial: Software Risk Management,* Barry Boehm, Ed. Washington, DC: IEEE Computer Press, 1988.
[37] U.S. Nuclear Regulatory Commission, *Fault Tree Handbook.* Washington, DC: U.S. Government Printing Office, 1986.
[38] R. T. Yeh, D. A. Naumann, R. T. Mittermeir, R. A. Schielmmer, W. S. Gilmore, G. E. Sumral, and J. T. Lebaron, "A commonsense management model," *IEEE Software,* vol. 8, pp. 23–33, 1991.

Clyde Chittister (M'82) is the Director of the Risk Program at the Software Engineering Institute (SEI) in Pittsburgh, Pennsylvania. The SEI is a federally funded research and development center sponsored by the U.S. Department of Defense (DoD) and operated by Carnegie Mellon University. The SEI mission is to provide leadership in advancing the state of the practice of software engineering to improve the quality of systems that depend on software. The objective of the Risk Program is to improve the management of risk in DoD programs involving software dependent systems. He has spent more than 20 years in both technical and management capacities in the real-time systems community. In industry, he held lead technical positions and program management positions for real-time process control systems. He also held senior management positions in both engineering and general management. Prior to joining the SEI he was general manager of a division of Brown Boveri Control Systems Inc. He joined the SEI in June 1985 and has held various positions at the Institute. His initial duties were in the Technology Transition function to define and implement the affiliates program with industry. He was then responsible for defining and implementing the Ada-Based Software Engineering Program and the Real-Time Systems Program, and served as program director for each program. More recently he led the task force that investigated and defined the concepts for the SEI Risk Program.

Dr. Chittister is a member of the IEEE Computer Society and the Society for Risk Analysis. He has been the program chair for the Constructive Cost Model (COCOMO) Users Group Conference for the last four years and has organized risk conferences with Aerospace Industry Associates (ALA) and the National Security Industry Associates (NSIA).

Yacov Y. Haimes (S'68–M'70–SM'76–F'80) received the B.S. degree in 1964 from the Hebrew University, Jerusalem, Israel. He received the M.S. degree in engineering in 1967 and the Ph.D. degree in engineering with distinction (majoring in large-scale systems engineering) in 1970 from the University of California at Los Angeles.

He is the Lawrence R. Quarles Professor of Engineering and Applied Science and Founding Director of the Center for Risk Management of Engineering Systems at the University of Virginia, Charlottesville, VA. He is a former Chairman of the Department of Systems Engineering at Case Western Reserve University, Cleveland, OH. His research, teaching and consulting activities are in decision-making under risk and uncertainty in large-scale systems within the hierarchical-multiobjective framework, motivated by his extensive studies of water resources systems. From September 1977 to August 1978, he was the American Geophysical Union Congressional Science Fellow and spent three months in the Executive Office of the President and eight months with the U.S. Congress. He is the author/co-author of five books, the most recent of which is *Hierarchical-Multiobjective Analysis of Large-Scale Systems*, published by Hemisphere Publishing Company, 1990.

Dr. Haimes is also the editor/co-editor of 15 other volumes, the author/co-author of more than 100 technical papers, Fellow of the IEEE and other societies, Vice President for Publications of the IEEE Systems, Man, and Cybernetics Society; Vice Chairman of the IFAC Systems Engineering Committee (1987-1990) and former Chairman of the Committee's Working Group on Water Resources Systems. He is an Associate Editor of *Automatica*; IEEE TRANSACTIONS ON SYSTEMS, MAN AND CYBERNETICS; *Risk Analysis*; and *Reliability Engineering and System Safety*.