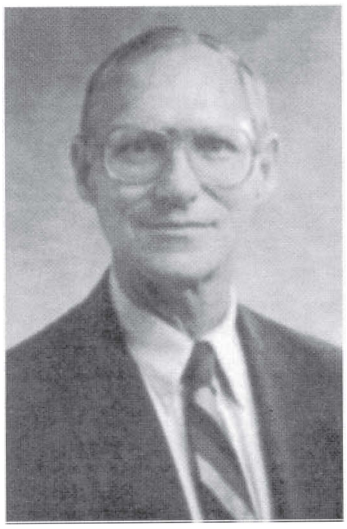
Software Economics: A Roadmap

Barry Boehm & Kevin Sullivan

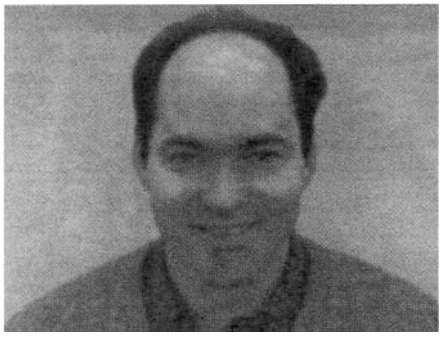
Key Research Pointers

- Principles, models, methods and tools for reasoning about and dynamic management of software development as an investment activity.
- Models for reasoning about benefits and opportunities in software development as well as costs and risks.
- Principles, models, methods and tools for dealing with uncertainty, incomplete knowledge, and market forces, including competition and change, in software development.
- Principles, models, methods, and tools for resolving multi-attribute decision issues in software design and development.
- Integration of economic considerations into software design and development methods.

The Authors



Barry W. Boehm is TRW Professor of Software Engineering and Director, Center for Software Engineering, University of Southern California. Barry Boehm received his B.A. degree from Harvard in 1957, and his M.S. and Ph.D. degrees from UCLA in 1961 and 1964, all in Mathematics. Between 1989 and 1992, he served within the U.S. Department of Defense (DoD) as Director of the DARPA Information Science and Technology Office, and as Director of the DDR&E Software and Computer Technology Office. He worked at TRW from 1973 to 1989, culminating as Chief Scientist of the Defense Systems Group, and at the Rand Corporation from 1959 to 1973, culminating as Head of the Information Sciences Department. He was a Programmer-Analyst at General Dynamics between 1955 and 1959. His current research interests focus on integrating a software system's process models, product models, property models, and success models via an approach called Model-Based (System) Architecting and Software Engineering (MBASE). His contributions to the field include the Constructive Cost Model (COCOMO), the Spiral Model of the software process, and the Theory W (win-win) approach to software management and requirements determination. He has served on the board of several scientific journals and as a member of the Governing Board of the IEEE Computer Society. He currently serves as Chair of the Board of Visitors for the CMU Software Engineering Institute. He is an AIAA Fellow, an ACM Fellow, an IEEE Fellow, and a member of the U.S. National Academy of Engineering.



Kevin Sullivan received his Ph.D. in Computer Science and Engineering from the University of Washington, Seattle, Washington, in 1994. He then joined the University of Virginia as Assistant Professor of Computer Science. He received a National Science Foundation Career Award in 1995; the first ACM Computer Science Professor of the Year Award from undergraduate students at the University of Virginia in 1998; and a University Teaching Fellowship in 1999. Sullivan is the author or the co-author of twenty four peer-reviewed journal and conference papers and three book chapters. He has been Principal Investigator or co-Principal Investigator on research grants totaling approximately $US1.8 million. Sullivan's research is on software-intensive systems, in general, and on software design, in particular. He has projects on modularity in software design, focusing on component integration and system evolution; the dependability of software-intensive systems; software economics; and reliability modeling analysis of computer-based systems. He also is or has been involved in multidisciplinary collaborations in reliability engineering, finance, international security, and radiation oncology.

Software Economics: A Roadmap

Barry W. Boehm
University of Southern California
Department of Computer Science
Los Angeles, CA 90089-0781 USA
+1 213 740 8163
boehm@sunset.usc.edu

Kevin J. Sullivan
University of Virginia
Thornton Hall
Department of Computer Science
Charlottesville, VA 22901 USA
+1 804 982 2206
sullivan@virginia.edu

ABSTRACT
The fundamental goal of all good design and engineering is to create maximal value added for any given investment. There are many dimensions in which value can be assessed, from monetary profit to the solution of social problems. The benefits sought are often domain-specific, yet the logic is the same: design is an *investment activity*. Software economics is the field that seeks to enable significant improvements in software design and engineering through economic reasoning about product, process, program, and portfolio and policy issues. We summarize the state of the art and identify shortfalls in existing knowledge. Past work focuses largely on costs, not on benefits, thus not on value added; nor are current technical software design criteria linked clearly to value creation. We present a roadmap for research emphasizing the need for a strategic investment approach to software engineering. We discuss how software economics can lead to fundamental improvements in software design and engineering, in theory and practice.

1 INTRODUCTION

The long-term exponential advance in computing and communications device capabilities is beginning to enable the incorporation of high-speed, low-cost, distributed information processing into technology components and system of all kinds, at all scales. This trend promises to provide enormous benefits by providing new functions and by improving the performance of existing functions. The potential for value creation is seen to be so great that it is driving information machinery into essentially all major social, business, and military human-machine systems: appliances, homes, communities, industries, research, design, armies.

Future of Sofware Engineering Limerick Ireland

Although hardware device innovation is the catalyst, it is software that embodies new value added functions. Software—broadly construed as any representation of abstract information content of a computing machine, whether encoded in fixed circuits or in the state of a mutable device—thus takes on a critical level of economic and social importance. This role is reflected in demand that far outstrips our production capacity [57], in world-wide expenditures on software now estimated at US$800 billion annually [13], and in many other aspects of the modern economy.

Yet, as the importance of software grows, its production and use remain among the most complex and problematical aspects of modern technology development. The symptoms are clear. One of many symptoms is that large projects fail at an alarming rate. The cost of failed projects has been estimated at $85 billion for U.S. business in 1998 alone, for example [16].

Project, program and business failures are inevitable, even desirable, in a dynamic marketplace. However, software development and use destroy value and create exposure to risks unpredictably and at an unacceptable rate. Doomed projects often consume considerable value before being cancelled. Software costs jump in ways inconsistent with expected risk, as exemplified by the experience of Garlan et al., in what appeared to be a straightforward integration task [30]. Delays lead to lost value, quality shortfalls, and missed opportunities. Unexpected absences of critical properties make costly systems unusable. Our inability to effectively manage the risk-return characteristics of software is a serious and difficult problem.

In this paper we trace many such difficulties to our failure to understand adequately the economics of software development and use, and thus our failure to make software and systems design decisions for products, processes, programs and portfolios that that are demonstrably consistent with the goal of maximizing value added. We discuss how a more sophisticated economic perspective on software design promises to improve the productivity of investments in software-intensive systems. We review the state of the art in software economics. We identify some

important shortcomings in the existing work on software economics. We then provide a roadmap for future research, and we discuss several current activities in that context.

2 THE NEED FOR RESEARCH

Software Engineering Decision-Making Today

Guided largely by the principle of separation of concerns, most software designers today make design decisions in an economics-independent "Flatland," where the focus is largely on representation structure and logical semantics. An analysis of sixteen books on software architecture and object-oriented design, for example, showed that only two included the word *cost* in the index. More generally, explicit links between technical issues and value creation appear not to be central concerns of most software engineers today. One part of the problem is that these links are not even understood very well in theory.

While software contributed primarily to off-line, backroom activities, designing in this Flatland was not particularly harmful. That is no longer the case. Software design decisions are now intimately coupled with fundamental business, public service, and other decisions in almost every field of endeavor. It is now essential that we understand how software design decisions relate to value creation in a given context.

Consider the business context. It is axiomatic in corporate finance that a publicly held firm measures value in monetary value in the marketplace, and that the primary goal of management is to maximize present value, incorporating expectations of future gains. Uncertainty, incomplete knowledge, and competition pose major challenges that demand intelligent investment strategies. Such an enterprise can create value in severeal ways: first, by producing or having options to produce benefits at a profit greater than that of competitors; second, by producing or having options to produce greater or different benefits at equal cost. Software design decisions in a business context must be linked to creating business value in these terms.

Less well known perhaps is that non-business enterprises, such as philanthropic foundations and universities, are also driven by maximal value creation objectives. For example, in "Philanthropy's new agenda: creating value," Porter and Kramer argue, "The goals of philanthropy may be different, but the underlying logic is still the same. Instead of competing in markets, foundations are in the business of contributing to society by using scarce philanthropic resources to their maximum potential. A foundation creates value when it achieves an equivalent social benefit with fewer dollars or creates greater social benefit for comparable cost" [59, p.126]. Similarly, in writing on strategic philanthropy, the President and Chief Executive Officer of the Pew Charitable Trust, says, "...trusts have begun to think more like venture capitalists, seeking to derive the greatest benefit from every strategic investment of capital, time, and talent—except, in Pew's case, the return on investment is measured not in profits but in long-lasting, positive, and powerful benefits to society" [64, pp. 230 – 231].

Software development involves costs, including time, talent and money. The benefits sought are measured in widely varying terms. Nevertheless, in all cases, the basic logic is the same. The goal is maximal value creation for a given investment. Understanding the relationships between technical properties and the decisions that produce them, on one hand, and value creation, on the other, is essential in world in which software is so important to all aspects of doing business or providing public services.

1 Software Engineering as a Value-Creation Activity

The core competency of software engineers is in making technical software product and process design decisions. Today, however, there is a "disconnect" between the decision criteria that tend to guide software engineers and the value creation criteria of organizations in which software is developed. It is not that technical criteria, such as information hiding architecture, documentation standards, software reuse, and the need for mathematical precision, are wrong. On average, they are enormously better than no sound criteria.

However, software engineers are usually not involved in or often do not understand enterprise-level value creation objectives. The connections between technical parameters and value creation are understood vaguely, if at all. There is rarely any real measurement or analysis of how software engineering investments contribute to value creation. And senior management often does not understand success criteria for software development or how investments at the technical level can contribute fundamentally to value creation. As a result, technical criteria tend to be applied in ways that in general are not connected to, and are thus usually not optimal for, value creation.

Software designers, engineers, and managers must begin to understand and reason effectively about the connections between technical decisions and enterprise-level value maximization. Understanding these connections will drive decision-makers at all levels to use better criteria, and to make better choices. One important adjustment is that decision-makers begin to think more strategically. Getting to this point requires that software specialists step out of "Flatland" and away from purely technical criteria that are not linked to enterprise-level value outcomes. The first step is to understand that the mismatch between the criteria that are used today and ones more aligned with value creation has several identifiable and remediable causes.

2 Sources of Technical-Value Mismatch

First, we lack adequate frameworks for modeling, measuring and analyzing the connections between technical

properties and decisions and value creation. Sullivan et al. have argued, for example, that central concepts in software engineering, such as information hiding [53], architecture [72], the spiral model [9], and heuristics on the timing of software design decisions, have yet to be linked adequately to business value, but that such linkages can be made. In particular, Sullivan et al. have argued that linkages can be established in terms of the *real options* value of the decision flexibility afforded by modular designs and phased project structures [76].

Consider phased project structures. They create embedded options to abandon or to redirect a project between phases, and thus to respond to changing conditions and the ongoing resolution of technical and market uncertainties. Such options can have significant value. Understanding options value can help inform project design because it can help the designer to decide when investing in options adds value.

Given an embedded option, perhaps obtained through an intentional investment, either holding it or exercising it can also be optimal for value in some cases. The failure to cancel projects quickly that new information shows are unlikely to succeed is a common example of not making a value-optimizing decision. The options view leads to a dynamic management view of projects, including decisions about whether and if so when to exercise options.

Another consequence of inadequately understood links is in conflicts among decision-makers, often in the form of arguments over whose technical criterion is better. Without links to value, there is little hope that such debates will converge, or that the decisions best for value will be taken.

Second, most software designers and engineers are not taught to reason about value creation as an objective or about how technical parameters can be manipulated for value creation purposes. Rather, technical measures tend to dominate pedagogy. Such measures are necessary but insufficient.

Third, the design space within which software designers operate today is inadequate. By design space we mean the set of technologies with which, and the organizational, regulatory, tax, market and other structures within which, software is developed and used. Designers are unable to make decisions that, if available, could significantly increase the value created by software development and use. Of course powerful new technologies have great value in improving software development productivity. However, beyond technology, the overall economic environment has shortcomings that need to be addressed. Examples include the inability of many firms to account for software as a capital investment; to access and exploit rich sets of third-party components; and to buy and sell software risk in the marketplace through warranties, insurance policies, and similar instruments.

Why an Increased Emphasis on Software Economics?

These and related issues fall in the category of software economics. The field of software economics is situated at intersection of information economics and software design and engineering. Its basic concern is to improve the value created by investments in software. It seeks to better understand relationships between economic objectives, constraints, and conditions, on one hand, and technical software issues, on the other, in order to improve value creation at all levels: project, program, portfolio, enterprise, industry, and national.

Software economics is not a new discipline, but there are several reasons why it should receive increasing attention. First, the end of the cold war, new technology, and globalization of capital markets have fundamentally altered the dynamics of technology innovation. The center has moved from large government projects to the commercial sector, where different measures of value apply and different dynamics exist, e.g., competition that makes time to market a critical success factor. Such factors must now be treated explicit in design decision-making.

Second, the impacts of software-enabled change today reach much further across and into organizations today than in the past [79]. Many aspects of an enterprise now have to be transformed for software-enabled change to create value. One example is order fulfillment for electronic retailing. Software systems are catalyzing great change, but complex human-machine systems with software as just a component must function for value to be created. Focusing on value creation demands a holistic perspective that considers all investments that have to be made for software investments to pay off. Without a holistic approach, inefficient investment patterns are likely to continue.

Third, there is an increasing understanding in business, philanthropy, government, and in most other major organizations, that value creation is the final arbiter of success for investments of scarce resources; and far greater sophistication than in the past is now evident in the search for value by the most effective organizations. In particular, there is a deeper understanding of the role of strategy in creating value. Strategic considerations dictate not only a holistic approach, but one that treats uncertainty, incomplete knowledge and competition in a sophisticated manner.

1 New Sources of Value

Along with a new emphasis on value and strategy is an increasing understanding that value is a complex, subtle idea. Consider the sophisticated ways in which markets value companies: not only for the profits that they might produce based on their current configurations, but also for strategic options that they have to reconfigure to exploit potential future opportunities and synergies [81]. Good strategists know that maximizing the value of an enterprise often depends on investing to create real options and

synergies. Increasingly strategy plays out in software design. Not for selling books at a profit has Jeff Bezos been named Time Magazine's 1999 Man of the Year.

The extraordinary market valuations of some internet companies reflects an assessment of the present value of uncertain future gains, including potential gains from the exercise of real options. The investment by Amazon.com in an infrastructure and an initial foray into books, for example, created not only a cash flow stream from book sales, but real options to enter other markets. The ability to exercise those options quickly depends in part on the ability to change the software on which the company runs, supported, in turn, by the architecture and other technical properties of the systems.

The "leap-frogging" of Intel and AMD in the race to keep the world's fastest microprocessor reflects the value of time in that market. The design processes have to be organized in part to enable effective competition. At a grander scale, when firms are driven to compete in this way the resulting increase in velocity of innovation and product range that occurs has a tremendous impact on technology and the economy.

Microsoft's independent-feature-based architectural style for many applications, combined with their synchronize-and-stabilize process, creates real options to abandon features late in development to meet time-to-market requirements [23]. In selecting and integrating product and process models, they are clearly doing so in a way that is meant to create value in the form of decision flexibility.

An important goal of modern software economics is thus to understand complex sources of value, and to clarify connections between technical and economic dimensions, including an explicit consideration of these higher-order terms. Without an understanding of how to reason about software design in these terms, it is unlikely that any prescriptive theory of software design will be adequate to the task of serving enterprise value-creation objectives as effectively as possible. Beyond the traditional issues of cost and schedule, it is now becoming important to address, in sound and systematic ways, such questions as whether the value of a portfolio of real option created by a given modular design is more than the cost of the investment in architecture or process needed to obtain it; and, of the possible portfolios corresponding to different modularizations, which is worth the most.

2 *New Measures of Value*

A complicating factor is that although value often should be measured in terms of money, this is not always true. Non-profit enterprises value non-monetary results. Some have argued that the greatest and most enduring of profit-making companies do not value money as the highest goal. Rather, they treat money as a critical enabler for creating value in other dimensions [20].

In some cases value is difficult to measure as a scalar quantity. Consider cost and public safety. These are two dimensions of value for which there is no simple, linear exchange formula. At the extremes of safety, it might incur tremendous costs to gain just a little more safety, a trade that might be judged uneconomical given other ways of using the same resources. Yet, when low on the safety scale, a small cost increment might give a disproportionate payoff in safety and be seen as very worthwhile [33].

In what dimensions and units is value measured? How are contingent future payoffs valued? What is the role of risk-aversion in valuing contingent payoffs? How should one reason about tradeoffs in multi-dimensional value spaces? How does one reason about such valuations in the face of uncertainty and incomplete knowledge. How does competition complicate models? A theory or practice of software engineering based on value criteria has to incorporate answers to these and many other related questions.

Answers to some such questions have been developed, mostly outside of the software engineering field. Decision (utility) theory [61] provides a framework for decisions under uncertainty in light of the risk aversion characteristics of the decision-maker. The mathematics of multi-objective decision-making has been addressed in depth [39]. *Smart Choices: A Practical Guide to Making Better Decisions,* is a good introduction for engineering decision makers [34].

Classical corporate finance is an extensive framework for making profit-oriented corporate investment decision-making in the face of uncertainty. The book of Brealey and Myers is a standard introduction [15]. Important topics include net present value (NPV) as an investment decision criterion; computing it by discounted cash flow analysis (DCF); and optimal portfolio theory, or how to invest in a portfolio of risky assets to maximize its return for a given level of risk. The NPV and DCF concepts are fundamental in building business cases, in general.

Work on real options [2,24,80,81] addresses major, often overlooked, shortcomings in DCF-based computations of the NPV of investment opportunities. DCF treats assets obtained by investing as passively held (like mortgages), not actively managed (like projects or portfolios). Yet, management often has the flexibility to make changes to real investments in light of new information. (e.g., to abandon a project, enter a new market, etc.) The key idea is to treat such flexibility as an option, and to see that in some cases such *real* (as opposed to financial) options can be priced using techniques related to those for financial (e.g., stock) options.

The fundamental advantage of the real options framework over the traditional DCF framework is that the resulting valuations incorporate the value added by making smart

choices over time. Options pricing is not the only available technique for valuing such decision flexibility. Teisberg presents perhaps the best available analysis of three key valuation techniques: options pricing, utility theory and *dynamic* discounted cash flow analysis. She explains the assumptions that each of these approaches requires as a condition of applicability, and the advantages and disadvantages of each [78].

The options pricing approach has two major advantages. First, it relieves the decision-maker of having to forecast cash flows and predict the probabilities of future states of nature. Second, it provides valuations that are based not on such subjective, questionable parameter values, but rather on data from the financial markets. The details are beyond the scope of this paper. In a nutshell, the decision-maker provides the current value of the asset under consideration and the variance in that value over time. That is enough to determine the "cone of uncertainty" in the future value of the asset, rooted at its current value and extending out over time as a function of the volatility.

The variance is obtained by identifying assets in the financial markets that are subject to the same risks as the one in question. A requirement for using this method for valuing decision flexibility is that the risk (variance) in the asset being considered be "in the span of the market," i.e., that it be a function of the risks in identifiable traded assets. The option to port a software system to a platform with an uncertain future might be valued this way, because the risk in the platform is arguably reflected in the behavior of the stock price of the company selling the platform. Because the market has already priced that risk, it has implicitly priced the risk in the asset under consideration, even if it is not traded. We get a *market-calibrated* price, rather than one based on subjective guesses. Much of the literature is vague on the need for spanning to hold. Amram and Kulatilaka provide a good introduction to this field [2].

The work of Baldwin and Clark is especially relevant. They view Parnas's information hiding modules [53] as creating options, which they then value as options (without depending on spanning). On the basis of this insight, they develop a theory of how modularity in design influenced the evolution of the industry structure for computers over the last forty years [6]. Sullivan et al., draw on this material and the material discussed above to sketch a more unified, options-based account of the value in software available through modularity, phased process models, and either delaying or accelerating key design decisions [76].

3 The Need for Multi-Stakeholder Satisficing

The question of valuation is clearly difficult. Another complicating factor is *who* is doing the valuing? Is it a company, a philanthropic foundation, a school or university, a government research funding agency? What does that person or entity value? Is it the likelihood of financial gain, the solution of major societal problems, the continuation of a valued culture, national security, or the pleasure of learning or of designing things that work?

Even the question *who* is often not simple. Any major software design activity involves many participants, each with its own goals and measures of value. Even if they agree on metrics—as in a coalition of profit-making companies cooperating to make a profit—they have conflicting interests in the distribution of gains. Reconciling economic conflicts of this kind is a key success factor in software development. Reconciliation has to be built into software processes, in particular.

A utilitarian view of this issue is possible. For a system to succeed in creating value for any one participant, it must create value for all whose contributions are critical to project success. The failure to satisfy any one critical party creates risks of compensatory actions that lead to a project failure, thus to the satisfaction of none. Ensuring a credible value proposition for each stakeholder at each point in time is thus an essential part of design. In practice, each player will carry a different amount and set of risks. Aligning rewards to, and dependencies of a project on, any given stakeholder has to account for their risks.

A non-utilitarian view of stakeholder reconciliation is also possible. Collins et al., discuss an approach based on a Rawlsian ethics of fairness [21,62]. The ideal is that the stakeholders in a given situation negotiate an arrangement under which each is treated fairly, where fairness is defined by fairness axioms (e.g., never cause more harm to the least advantaged stakeholder), and each player negotiates as if it were unaware of its self-interest. Collins et al. present a fictional scenario involving a software provider, buyer, users, and a "penumbra" of people who are affected by the software. The hospital is the buyer, the doctors, nurses and others the users, and patients, who might be harmed by wrong dosages, are the penumbra.

An analogous situation that is becoming quite visible at the national policy level in the United States relates to private ownership of critical civic infrastructures. Most of these systems have come to depend on software and information systems in ways that most people never imagine [41,77]. There is great concern that many of these systems are no longer adequately dependable, given both our increased reliance on them and that they are now operating in environments that are significantly more threatening than those for which they were designed. They have been opened to manipulation through networks, outsourcing of code development, and other means, and are vulnerable to the growing capabilities of potential adversaries. Is public interest in the dependability of transportation and logistics, banking and finance, energy production, transmission and distribution, and other such infrastructures perfectly aligned with the interests of the shareholders of the private firms that own these infrastructures?

Understanding how to integrate value considerations into complex, software-intensive development processes is a software design and engineering challenge, where software design is construed in broad terms, to include public policy. The stakeholder win-win concept and its integration into the Win-Win Spiral Lifecycle Model [10] represent a serious attempt at economics-driven software design. The design of this process model is clearly responsive to, and indeed based on, a consideration of the economics of complex software development processes. It also provides a way to embed Rawlsian ethical considerations into the daily practice of software engineers [26].

From Win-Win, it is a relatively easy mental jump to related models based on strategy in multi-player games. Tit-for-Tat is an effective strategy, for example, in a two-player, iterated prisoner's dilemma (IPD) context. The IPD is an abstract model that captures aspects of many business interactions. In this game, win-win occurs if each side cooperates. In this case, each makes a small gain. Lose-lose occurs if both sides defect. The penalty to each side is large. The interesting part is that in win-lose, where one side cooperates but the other defects, the winner gets a large payoff, but the penalty to the loser is significant. Thus there is a short term incentive to defect, but a long term need for cooperation.

Value creation in this world depends on whether you're likely to encounter the other player for another round in the future. If not, defecting is a reasonable strategy. If so, cooperating is reasonable because cooperation is the only strategy likely to produce consistent positive returns over time. However, cooperating naively with a consistent defector, e.g., one who takes your money but provides an unreliable product, is clearly not optimal. Over time, limited retaliatory defection—i.e., tit-for-tat—has been found to be a highly competitive strategy. It punishes defections in a limited way to deter future defections but is otherwise willing to cooperate [5]. Software design in a world of dynamically assembled profit-making virtual enterprises might well be subject to such economic considerations, as might the design of automated agent-based electronic commerce capabilities.

4 Future Trends Create Additional Challenges

Future trends will continue to exacerbate this situation. The world is changing rapidly in ways that make the situation ever more challenging. While ever smaller, less costly devices penetrate into the technology fabric, the World-Wide Web and Internet have the potential to connect everything with everything. Autonomous agents making deals in cyberspace will create a potential for chaos. Systems of systems, networks of networks, and agents of agents will create huge intellectual control problems.

Further, the economics of software development leave system designers with no choice but to use large commercial-off-the-shelf (COTS) components in their systems. Developers have no way of knowing precisely what is inside of these COTS components, and they usually have very limited or no influence over their evolutionary paths.

The PITAC Report accurately states [57, p.8] that "The IT industry expends the bulk of its resources, both financial and human, in rapidly bringing products to market." The dizzying pace of change continues to increase. Software architecture and COTS decisions are made in great haste. If you marry an IT architecture in haste, you no longer even have the opportunity to repent at leisure. Commercial companies with minimal electronic commerce capabilities must now adapt to e-commerce or die.

Of course, these trends also make this a time of fantastic opportunity. The PITAC Report is "right on" in emphasizing that IT offers us the potential to significantly improve our quality of life by transforming the ways we learn, work, communicate, and carry out commerce, health care, research, and government. Value creation opportunities abound, but the path "from concept to cash" [79] is becoming ever more treacherous.

A new focus on software economics is needed. We now discuss the history and the current status of software economics, with the goal of understanding how it should evolve to be better positioned to address important emerging issues in software design.

History and Current Status of Software Economics

Software economics can be considered as a branch of information economics, a subfield of economics which began to receive serious treatment in the 1960's. Its original subjects were such topics as the economics of advertising and search for best prices [74], the economics of investments in research and development [3], and the economics of the overall knowledge industry [45]. A good early comprehensive treatment of information economics is *Economic Theory of Teams* [48].

The first comprehensive application to computing issues was Sharpe's *The Economics of Computers* [71]. It covered such issues as choices between buying, leasing, or renting computer systems; pricing computer services, and economies of scale in computer systems. It had a small section on software costs, based largely on the first major study of this topic, performed by System Development Corporation (SDC) for the U.S. Air Force [52].

The SDC study formulated a linear regression model for estimating software costs. Although it was not very accurate, it stimulated considerable research into better forms for software cost models in the 1970's and early 1980's. This resulted in a number of viable models still in use today, such as SLIM [60], PRICE S [29], COCOMO [8], SEER [37], Estimacs [66], and SPQR/Checkpoint [38].

Besides the COCOMO model, *Software Engineering*

Economics [8] contained a summary of the major concepts and techniques of microeconomics (production functions, economies of scale, net value, marginal analysis, present value, statistical decision theory), with examples and techniques for applying them to software decision situations. Related contemporary works were the monumental *Data Processing Technology and Economics* [56], a detailed compendium of cost estimating relationships and cost comparisons for computing equipment and services (unfortunately with a short half-life); *Computers and Profits* [40], applying information-economics techniques to computer-related decisions; and *The Economics of Computers: Costs, Benefits, Policies and Strategies* [31], providing economic techniques for managing computer centers and for related purchasing and strategic-management decisions.

A number of mainstream software engineering techniques implicitly embody economic considerations. Software risk management uses statistical decision theory principles to address such questions as "how much (prototyping, testing, formal verification, etc.) is enough?" in terms of buying information to reduce risk. Spiral, iterative, and evolutionary development models use risk and product value considerations to sequence increments of capability. The spiral model's "Determine Objectives, Alternatives, and Constraints" step [9] was adapted from RAND-style treatments of defense economic analysis [35].

Parnas's notion of *design for change*, is based on the recognition that much of the total lifecycle cost of a system is incurred in evolution, and that a system that is not designed for evolution will incur tremendous cost [54]. However, the work focuses on modularity as a structural issue, per se, more than on the weighing of lifecycle costs, benefits and value creation. The over-focus on structural issues has carried through much of the more recent work on software architecture [72].

Architecture and economics also play a large role in dealing with software reuse. Some good books in this are [36,43,58,63]. Economics concepts of satisficing among multi-stakeholder criteria and utility functions as articulated in Simon's *The Sciences of the Artificial* [73] have been incorporated in software engineering approaches such as Participatory Design, Joint Application Design, and stakeholder win-win requirements engineering [11,17].

Shortcomings that Need to be Addressed

Currently, our ability to reason about software cost is considerably stronger than our ability to reason about software benefits, or about such benefit sources as development cycle time, delivered quality, synergies among concurrent and sequential projects, and real options, including strategic opportunities. The trends toward software-based systems discussed above make it clear that the ability to reason about both costs and benefits, sometimes in sophisticated terms, and under such difficulties as uncertainty, incomplete information, and competition, will be a critical success factor for future enterprises.

A good example is Rapid Application Development (RAD). As discussed above, the US PITAC Report [57] accurately states that the information technology industry focuses on rapidly bringing products to market. However, most software cost and schedule estimation models are calibrated to a minimal cost strategy, which is not always (and increasingly not) equivalent to a value maximization strategy. Each such approach has an estimation model similar to a classic schedule estimation rule of thumb:

$$\text{Calendar Months} = 3 * \sqrt[3]{(\text{Person-Months})}.$$

Thus, if one has a 27 person-month project, the most cost-efficient schedule would be $3 * \sqrt[3]{(27)} = 9$ months, with an average staff size of 3 people. However, this model captures only the direct cost of the resources required to develop the project. It completely fails to account for the opportunity cost of delay in shipping a product into a competitive marketplace, which is often decisive today.

A product in development can be viewed as a *real option* on a market, like an option on a stock. Shipping the product to market is the analog of exercising the option. The entry of a competitor into a market, taking away a share of the cash flow stream that could otherwise be exploited, is the analog of a sharp, discrete drop in the stock price, i.e., of a dividend. It is known that for stocks that do not pay dividends, waiting to exercise is optimal. however, waiting to own a stock that pays dividends, or to enter a market that is subject to competitive entry, incurs an opportunity cost: only the owner of the stock (or market position) gets the dividend (profits). Thus, dividends (or the threats of competitive entry) create incentives to exercise early.

Here we have a rigorous economic explanation for time-to-market pressure. Understanding such issues is critical to optimal software design decision making, where design decisions include such decisions as that to "ship code."

If time-to-market is critical, a solution more attractive than that suggested by the rule of thumb above would involve an average of 5.2 people for 5.2 months, or even 6 people for 4.5 months. The earlier work assumes a non-competitive environment, reflecting its orientation to government contracts and classical batch-processing business systems. The recent COCOMO II model [14] has an emerging extension called CORADMO to support reasoning about rapid schedules for smaller projects.

Not only are better software development estimation models needed, but also they need to be integrated with counterpart models in business operational mission domains, in order to reason about tradeoffs between software development time, function, quality, and the

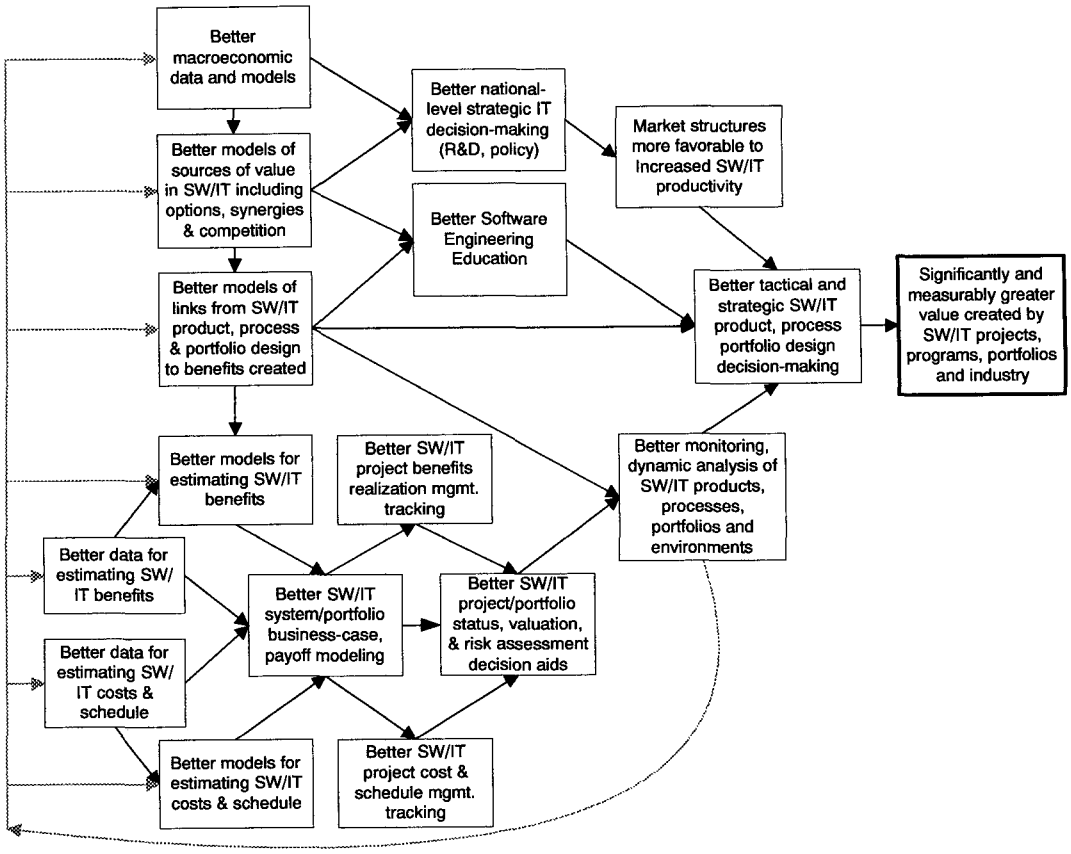


Figure 1: Roadmap for research in software engineering economics.

ability to create value. Of particular importance is the need for work on software economics to move from static notions of (uncertain) costs and benefits to dynamic and strategic concepts of value creation through flexible decision making under highly demanding circumstances over time. These needs are discussed further below.

3 SOFTWARE ECONOMICS ROADMAP

Our roadmap for the next major phase of research in software economics begins with the goal of developing fundamental knowledge that will enable significant, measurable increase in the value created over time by software and information technology projects, products, portfolios and the industry.

Working backwards from the end objective, we identify a network of important intermediate outcomes. The roadmap in Figure 1 illustrates these intermediate outcomes, dependence relationships among them, and important feedback paths by which models and analysis methods will be improved over time. The lower left part of the diagram captures tactical concerns, such as improving cost estimation for software projects, while the upper part captures strategic concerns, such as reasoning about real options and synergies between project and program elements of larger portfolios.

Making Decisions that are Better for Value Creation

The goal of our roadmap is supported by a key intermediate outcome: designers at all levels must make design decisions that are better for value added than those they make today. Design decisions are of the essence in product and process design, the structure and dynamic management of larger programs, the distribution of programs in a portfolio of strategic initiatives, and to national software policy. Better decision-making is the key enabler of greater value added.

Design decision-making depends in turn on a set of other advances. First, the design space within which designers operate needs to be sufficiently rich. To some extent, the design space is determined by the technology market structure: what firms exist and what they produce. That structure is influenced, in turn, by a number of factors, including but not limited to national-level strategic decision-making, e.g., on long-term R&D investment policy, on anti-trust, and so forth. The market structure determines the materials that are produced that designers can then employ, and their properties.

Second, as a field we need to understand better the links between technical design mechanisms (e.g., architecture), context, and value creation, to enable both better education and decision-making in any given situation. An improved understanding of these links depends on developing better models of sources of value that are available to be exploited by software designers in the first place (e.g., real options).

Third, people involved in decision-making have to be educated in how to employ technical means more effectively to create value. In particular, they personally need to have a better understanding of the sources of value to be exploited and the links between technical decisions and the capture of value.

Fourth, dynamic monitoring and control mechanisms are needed to better guide decision-makers through the design space in search of value added over time. These mechanisms have to be based on models of links between technical design and value and on system-specific models and databases that capture system status, valuation, risk, and so on: not solely as functions of endogenous parameters, such as software development cost drivers, but also of any relevant exogenous parameters, such as the price of memory, competitor behavior, macroeconomic conditions, etc.

These system-specific models are based on better cost and payoff models and estimation and tracking capabilities, at the center of which is a business-case model for a given project, program or portfolio. We now discuss some of the central elements of this roadmap in more detail.

Richer Design Spaces

The space in which software designers operate today is inadequate. One of the important reasons for this is that the market structures within which software development occurs are still primitive in comparison to those supporting other industries. We are less able to build systems from specialized, efficiently produced, volume-priced third-party components than is possible in many other fields. We are also less able to use markets to manage risk through warranties, liability insurance, etc., than is common in most fields. The inability to manage risk by the use of market mechanisms is a major hindrance to efficient production.

Links Between Technical Parameters and Value

Software design involves both technical and managerial decisions. The use of formal methods or the shape of an architecture are technical issues. The continuation or reorientation of a program in light of new information is managerial. The two are not entirely separable. The selection of a life-cycle model is a technical decision about the managerial framework for a system. Moreover, even where software engineering is concerned with technical issues, the connection to value creation is what matters.

The promotion of Parnas's concept of information hiding modules, for example, is based on the following rationale: most of the life-cycle cost of a software system is expended in change [42]. For a system to create value, the cost of an increment should be proportional to the benefits delivered; but if a system has not been designed for change, the costs will be disproportionate to the benefits [53]. Information hiding modularity is a key to design for change.

Design for change is thus promoted as a value-maximizing strategy provided one can anticipate changes correctly. While this is a powerful heuristic, we lack adequate models of the connections between this technical concept and value creation under given circumstances. What is the relationship between information hiding modularity and design interval? Should one design for change if doing so takes any additional time in an extremely competitive marketplace in which speed to market is a make-or-break issue? Is information hiding obligatory if the opportunity cost of delay might be enormous? What is performance if of the essence? How does the payoff from changing the system relate to the cost of enabling the change? What role does the timing of the change play? What if it is not likely to occur until far in the future? What if the change cannot be anticipated with certainty, but only with some degree of likelihood? What if the change is somewhat unlikely to be needed but in the case that it is needed, the payoff would be great [76]? Value-optimal technical design choices depends on many such factors.

Similarly, early advocates of the aggressive use of formal methods promoted them on the grounds that software could not be made adequately reliable using only informal and ad hoc methods, but only through the use of formal methods. Some thought that systems that could not be proven right should not be built. The implied hypothesis (all too often promoted as fact) was that using formal methods was optimal for value, if only because value simply could not be created, net of cost and risk, otherwise.

Subsequent experience has shown that hypothesis to have been wildly incorrect. In particular, it has turned out to be possible to create tremendous value without formal methods. Some early advocates have admitted that and pose interesting questions about why things turned out this way. One answer is that the assumed links were based on a view of software products as relatively unchanging that turned out not to be an accurate view.

We are not saying that formal methods cannot add value. They obviously can in some circumstances: e.g., for high-volume, unchanging artifacts, such as automotive steering-gear firmware. We still do not understand adequately the economic parameters under which investments in the use of formal methods create value. Recent work, e.g., of Praxis, Inc., is improving our understanding. Serious attempts to support limited but significant formal methods in industrial, object-oriented design modeling frameworks, such the Catalysis variant of UML [25], should provide additional information over time.

Links Between Software Economics and Policy

Understanding technology-to-value links is critical to making smart choices, not only at the tactical project level, but also in strategic policy-making: e.g., in deciding whether to promote certain results as having demonstrated value, and in selecting research activities having significant potential to achieve long-term, strategic value creation objectives. Whereas software engineering is about making smart choices about the use of software product and process technologies to create value, software engineering research policy is about making smart choices about how to change the software engineering design space to enable greater value creation over time.

The question of who decides precisely what form of value research is to seek, and what value the public is getting for its investment in research and development, is a deep question in public policy. Without trying to answer it fully, we summarize some current trends and provide a framework for software research and development investment policy research.

The prevailing definition of the value to be created by public investment in research has changed in significant ways over the last decade. That change is one of the factors that demands that greater attention now be paid to software economics. During the Cold War and prior to the globalization of commerce and the explosion of advanced technology development in the commercial sector, the nation's R&D investments were driven largely by national security concerns. Value creation meant contributing to that mission. Today, many argue, the concern has shifted dramatically to economic competitiveness. R&D investments in pre-competitive technologies are meant to pay off in design space changes that enable industries to create and societies to capture greater economic value.

In the United States, a major strategic emphasis has been put on new public investment in R&D in information technology, with software, broadly construed, as a top priority. This emphasis is justified on several grounds. First, society is coming to rely on systems that depend on fragile, unreliable, and insecure software. Second, our ability to produce software that is both powerful and easy enough to use to create value is inadequate. Third, our capacity to produce the amount of software needed by industry is inadequate. There are thus two basic dimensions of value in the calls for new public investments in software R&D: public welfare, and economic prosperity. Realizing value in these dimensions is as much a concern of the public as profit is for shareholders.

Software R&D Investment Policy Research Framework

At the level of corporate and national strategic software R&D investment policy, the question, then, is what portfolio of investments—in larger programs and individual projects—is needed to deliver the returns desired over time at selected risk levels? (Risk is a measure of the variance on future returns viewed as a random variable.) The returns will occur in short, intermediate, and long time-frames. How can a portfolio be managed for maximal value creation in the indicated dimensions? How can the return on resources investment be evaluated? Should individual projects be evaluated for success or failure?

Individual projects are risky. There is often too little information to value them precisely. Rather, funding a project can be seen as producing an option to make a follow-on investment in the next stage, contingent on success. Over time, as research generates new information, the option value fluctuates. At the end of the phase a decision is made on whether to exercise the option to invest in the next phase. Declining to invest is not a signal that the researcher failed or that the initial investment was misguided, only that in light of current information, it would not be optimal to invest in the next phase.

The staged investment approach permits concepts to be dropped or changed if they turn out not to work, or to be promoted through increasing levels of commitment. The corporation or society benefit when a transition to profitable production is made and where *aggregate* profits more than compensate for the investment in the research *portfolio*.

It is critical that individual research efforts not be judged as a matter of policy—prospectively or retrospectively—in terms of potential or actual contribution to value realized by the society or corporation. That would drive all research to the short term. The return-on-investment calculation should occur at the program and portfolio level. For example, foundation funding for what was seen at the time as far-out research in biology catalyzed the green revolution. The program ran about fifteen years. Not every project within such a program succeeds, nor do most successful projects directly ameliorate hunger. Rather, successful projects contribute to a complex of intermediate results that lead to end results that, when transitioned into production, ultimately produce the benefits for society. The return is weighed against the investment in the overall program.

For basic research in software it is similarly essential to insist that individual projects not be evaluated solely in terms of actual or potential *direct* payoff to society or business. At the same time, one must insist that strategic programs show payoffs over sufficiently long time frames. Individual projects can be evaluated prospectively or in terms of their potential to contributed intermediate results that could further a strategic program, and retrospectively in terms of whether they did so. One must be determined to redirect or abandon software research programs if they do not deliver realized benefits to a corporation or society over sufficient periods. Software economics thus includes the economics of investments in creating new knowledge about how to produce software.

Finally, strategy is multi-dimensional. Realizing the benefits of investments in the creation of knowledge through basic research is unlikely if too few people are taught about it. The education of software and information technology designers who occupy technical and business positions will play a significant role in realizing economic benefits of research in software, in general, and of research in software economics, in particular. Garnering the benefits of better design spaces and software technologies and investment models depends on knowledgeable professional experts using them effectively.

Better Monitoring & Control for Dynamic Investment Management

Software-intensive systems design generally occurs in a situation of uncertainty and limited knowledge. Designers are confronted with uncertainties about competitors, technology development, properties of products, macro-economic conditions, the status of larger projects within which a given activity is embedded. Conditions change and new information is gained continuously. The benefits that were envisioned at the beginning of such a project often turn out to be not the ones that are ultimately realized, nor are the paths by which such activities progress the ones that were planned. Complex projects take complex paths. The search for value in spaces that are at best only partially known is necessarily dynamic if it is to be most effective.

Beyond a better understanding of software design as a decision-making process, a better design space in which to operate, a better understanding of the context-dependent linkages between technical properties and value creation, and better educated decision-makers, software designers need mechanisms to help them navigate complex situations in a manner dynamically responsive to new information and changing conditions. We need models for both the systems being developed and for decision processes that support dynamic monitoring and control of complex software development activities. Dynamic management of investment in the face of significant uncertainties and gaps in knowledge is critical at levels from the single project to corporate and national software R&D investment policy.

Multiple models of several kinds will be used at once in any complex program. Models will be needed to guide and to support monitoring and control in the areas of product (e.g., architecture, verification), process (e.g., overall lifecycle), property (e.g., dependability), costs (e.g., for staff, materials, overhead), risk (e.g., lawsuits, liability judgements, failure due to technical or managerial difficulties), opportunities (e.g., to improve a product, to extend it to exploit new markets or other sources of value, or to follow with a synergistic new function), major programs (e.g., the dependencies among projects that determine ultimate success), corporate or national portfolios (constituent projects and how they support strategic objectives), uncertainty (e.g., project risks within programs and co-variance properties), markets (resources, needs, competition), etc.

Models at all of these levels are relevant to technical software design decision-making. Product architectural design decisions, for example, are critical to determining strategic opportunities and in mitigating technical and other risks. Such models and associated dynamic decision processes should be developed, integrated into software design activities, and related to our existing software design decision criteria. To enable the use of such models in practice, tool and environment support will often be needed.

4 IMPROVING SOFTWARE ECONOMICS WITHIN AN ENTERPRISE

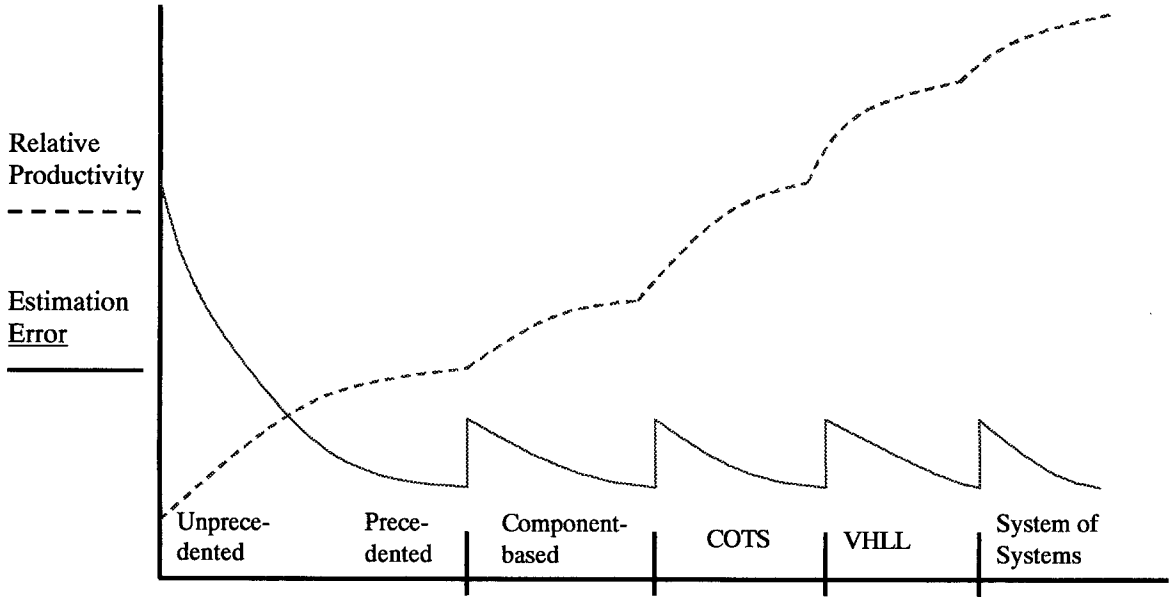
The lower portion of the roadmap in Figure 1 summarizes a closed-loop feedback process for improving software economics within an enterprise. It involves using better data to produce better estimates of the likely costs and benefits involved in creating, sustaining, and employing a portfolio of software and information technology assets. These estimates can be used to initiate a dynamic management process in which progress toward achieving benefits is tracked with respect to expenditure of costs, and corrective action is applied when shortfalls or new opportunities arise. This tracking also results in more relevant and up-to-date data for improving the cost and benefit estimation models for use in the next round of the firm's initiatives. In this section, we discuss three key components of this process: modeling costs, benefits, and value; tracking and managing for value; and design for lifecycle value.

Modeling Costs, Benefits, and Value

1 Modeling Software Development Cost, Schedule, and Quality

In Section 2 we discussed several software cost estimation models, and indicated that each had at least passed a market test for value by remaining economically viable over at least a decade. Their relative accuracy remains a difficult question to answer, as data on software cost, schedule, and quality is far from uniformly defined. A significant step forward was made with the core software metric definitions developed in [67], but there is still about a ±15% range of variation between projects and organizations due to the counting rules for data. Example sources of variation are the job classifications considered to be directly charging to a software project, the way an organization counts overtime, and the rules for distinguishing a defect from a feature.

Figure 2. Productivity and Estimation Accuracy Trends



This has led to a situation in which models calibrated to a single organization's consistently collected data are more accurate than general-purpose cost-schedule-quality estimation models. Some particularly good examples of this in the software quality and reliability estimation area have been AT&T/Lucent [50], IBM [18], Hewlett Packard [32], the NASA/CSC/University of Maryland Software Engineering Lab [49], and Advanced Information Services [28].

The proliferation of new processes and new technologies is another source of variation that limits the predictive accuracy of estimation models. For example, it required 161 carefully-collected data points for the calibration of COCOMO II [14] to reach the same level of predictive accuracy (within 30% of the actuals, 75% of the time) that was reached by the original COCOMO model [19] with 63 carefully-collected Waterfall-model data points [8].

Alternative estimation approaches have been developed, such as expertise-based, dynamics-based, case-based, and neural net models; see [13] for further details. Neural net and case-based models are still relatively immature. Dynamic models are particularly good for reasoning about development schedule and about adaptation to in-process change [1,44], but are hard to calibrate. Expertise-based methods are good for addressing new or rapidly changing situations, but are inefficient for performing extensive tradeoff or sensitivity analyses. All of the approaches share the difficulties of coping with imprecise data and with changing technologies and processes.

2 The Elusive Nature of Software Estimation Accuracy

In principle, one would expect that an organization could converge uniformly toward perfection in understanding its software applications and accurately estimating their cost, schedule, and quality. However, as the organization better understands its applications, it is also able to develop better software development methods and technology. This is good for productivity and quality, but it makes the previous estimation models somewhat obsolete. This phenomenon is summarized in Figure 2.

As an organization's applications become more precedented, its productivity increases and its estimation error decreases. However, at some point, its domain knowledge will be sufficient to develop and apply reusable components. These will enable a significant new boost in productivity, but will also increase estimation error until the estimation models have enough data to be recalibrated to the new situation. As indicated in Figure 2, a similar scenario plays itself out as increased domain understanding enables the use of commercial-off-the-shelf (COTS) components and very high level languages (VHLL). A further estimation challenge arises when the organization becomes sufficiently mature to develop systems of systems which may have evolved within different domains.

3 Modeling Benefits and Value

We were careful not to put any units on the "productivity"

scale in Figure 2. Measuring software productivity has been a difficult and controversial topic for a long time. Great debates have been held on whether source lines of code or function points are better for measuring productivity per person-month. Basically, if your organization has the option of developing software at different language levels (assembly language, 3GL, 4GL), function points will be preferable for measuring productivity gains, as they are insensitive to the extra work required to produce the same product in a lower-level language. (However, for the same reason, source lines of code will be preferable for estimating software costs.) If your organization develops all its software at the same language level, either is equally effective.

However, it is not clear that either size measure is a good proxy for bottom-line organizational productivity. One problem is behavioral, and can be summarized in the acronym WYMIWYG (what you measure is what you get). In a classic experiment, Weinberg gave the same programming assignment to several individuals, and asked each to optimize a different characteristic (completion speed, number of source statements, amount of memory used, program clarity, and output clarity). Each individual finished first (or in one case, tied for first) on the characteristic they were asked to optimize [82]. The individual asked to optimize completion speed did so, but finished last in program clarity, fourth out of five in number of statements and memory used, and third in output clarity. A thorough treatment of this and other risks of "measurement dysfunction" is provided in [4].

The second problem is that it is not clear that program size in any dimension is a good proxy for organizational productivity or value added. The popular design heuristic KISS (keep it simple, stupid) would certainly indicate otherwise in many situations. This leads us again to the challenge of modeling the benefits and value of creating a software product.

In contrast to methods for modeling software costs, effective methods for modeling software benefits tend to be highly domain-specific. The benefits of fast response time will be both modeled and valued differently between a stock exchange, an automobile factory, and a farm, just because of the differences in time value of information in the three domains.

4 General Benefit-Modeling Techniques

However, there are more general techniques for modeling the contribution of a software product to an organization's benefits. These frequently take the form of a causal chain linking the organization's goals and objectives to the development or acquisition of software. Examples are Quality Function Deployment [27], Goal-Question-Metric [7], and the military Strategy-to-Task approach.

A significant recent advance in this area is the Results Chain used in the DMR Benefits Realization Approach (DMR-BRA) [79]. As shown in Figure 3, it establishes a framework linking Initiatives which consume resources (e.g., implement a new order entry system for sales) to Contributions (not delivered systems, but their effects on existing operations) and Outcomes, which may lead either to further contributions or to added value (e.g., increased sales). A particularly important contribution of the Results Chain is the link to Assumptions, which condition the realization of the Outcomes. Thus, in Figure 3, if order to delivery time turns out not to be an important buying criterion for the product being sold, the reduced time to deliver the product will not result in increased sales.

This framework is valuable not only for evaluating the net value or return on investment of alternative initiatives, but also in tracking the progress both in delivering systems and contributions, and in satisfying the assumptions and realizing desired value. We will return to the Benefits Realization Approach below.

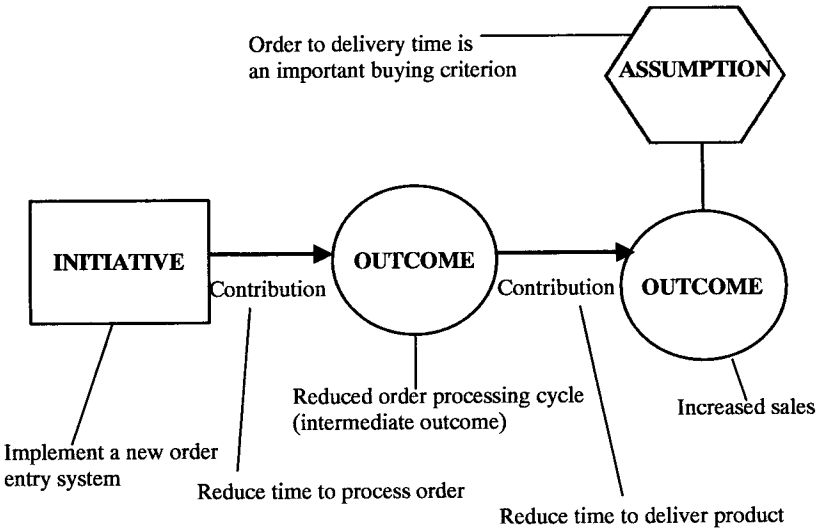
5 Modeling Value: Relating Benefits to Costs

In some cases, where benefits are measured in terms of cost avoidance and the situation is not highly dynamic, one can effectively apply net present value techniques. A good example in the software domain deals with investments in software product lines and reusable components. Several useful models of software reuse economics have been developed, including effects of present value [22] and also reusable component half-life [46]. An excellent compendium of economic factors in software reuse is [58].

Even with software reuse, however, the primary value realized may not be in cost avoidance but rather in reduced time to market, in which case the value model must account for the differential benefit flows of earlier or later market penetration. Some organizations in established competitive marketplaces (e.g., telecommunications products) have quite sophisticated (and generally proprietary) models of the sensitivity of market share to time of market introduction. In other domains, such as entrepreneurial new ventures, models relating market share to time of market introduction are generally more difficult to formulate.

Another major challenge in modeling costs, benefits, and value is the need to deal with uncertainty and risk. Example sources of uncertainty are market demand, need priorities of critical stakeholders or early adopters, price, macro-economic conditions (e.g., shrinking markets in Asia or Latin America), technology unknowns, competitor unknowns, and supply scarcities. These uncertainties have spawned an additional sector of the economy which performs consumer surveys, market projections, technology evaluations, etc., and sells them to organizations willing to buy information to reduce risk.

Figure 3. Benefits Realization Approach Results Chain



Tracking and Managing for Value

A good indicator of the current status and trends in models for software project tracking and managing is provided by the related key practices in the Software Engineering Institute (SEI) Capability Maturity Model for software Version 1.1 [68,55], and in the recent draft CMM-Integrated-Systems/Software Engineering (CMMI-SE/SW) Version 0.2b [69].

In the software CMM Version 1.1, the basic Key Process Area, situated at Level 2, is called Software Project Tracking and Oversight. It has 13 Activities-Performed elements, which include tracking to and updating a documented plan; reviewing and controlling commitments; tracking and taking necessary corrective actions with respect to software size, effort, cost, computer resources, schedule, and risks; recording data; and performing formal and periodic internal project reviews.

This framework is a sound implementation of the fundamental project management feedback process of monitoring progress and resource expenditures with respect to plans; and of performing corrective actions, including appropriate plan revisions, where necessary. It is relatively advanced with respect to risk management. However, it is very narrowly focused on the software artifacts to be produced, and not much on their contribution to achieving benefits or organizational goals.

The corresponding Level 2 process in the CMMI-SE/SW is called Project Monitoring and Control. Its implementation of the fundamental project management feedback process is roughly the same, including the emphasis on risk management, which is also a separate Process Area in the CMMI-SE/SW. It is more system-oriented, in that the product and task attributes to be monitored and controlled include such attributes as size, complexity, weight, form, fit, and function. However, except for possible broad interpretations of "form, fit, and function," it is also very narrowly focused on the artifacts to be produced, and not much on their contribution to achieving benefits or organizational goals.

Both the software CMM and the CMMI-SE/SW have Level 4 Process Areas which relate more to customer and organizational needs and goals. Both use a relatively advanced definition of "quality" with respect to such traditional measures of software quality as delivered defect density. In the software CMM, a primary activity involves understanding the quality needs of the organization, customer, and user, exemplified by the use of surveys and product evaluations. In the CMMI-SE/SW, particular quality needs are additionally exemplified such as functionality, reliability, maintainability, usability, cycle time, predictability, timeliness, and accuracy. The CMMI also emphasizes traceability to not only to requirements but also to business objectives, customer discussions, and market surveys.

Again, these are quite advanced in their focus on customer needs and business objectives, but their primary focus remains on tracking and managing the execution of the project rather than on the value it will presumably deliver. Concepts such as a business case which validates the economic feasibility of the project, and which serves as a basis for tracking the continuing validity of assumptions underlying the project's business case are not explicitly mentioned. In practice, the usual "earned value" system

used to track progress vs. expenditures uses the budgets for the project tasks to be performed as the value to be earned, rather than the expected business value associated with the product's operational implementation.

Opportunities for Improvement

In the context of our previous discussions of value creation via information technology, the current normative tracking and managing practices as exemplified by the CMM's leave open a number of opportunities for improvement. These include improvements in the nature of the achievements to be monitored and controlled, and improvements in the nature of the corrective actions to be performed in case of shortfalls in the projected achievements.

Improvements in the nature of the achievements to be monitored and controlled have been discussed in the context of dynamic investment management in Section 3, and will be discussed further in Section 5. A particularly attractive initial improvement to address is the application of business-value concepts to traditional earned value systems. One approach would be to use the project's business case analysis as the basis of accumulating projected business value, rather than (or in addition to) the current measure of success in terms of task-achievement based value.

Improvements in the nature of corrective actions can involve reorganization of the project's process and the system's architecture to best support adaptation to dynamic business objectives. If an entrepreneurial startup's primary objective, for example, is to demonstrate a competitive agent-based electronic commerce system at COMDEX in 9 months, the driving constraint is the inflexible date of COMDEX in 9 months.

An appropriate reorganization of the process and architecture involves using a Schedule as Independent Variable (SAIV) process model, in which product features are dropped in order to meet the 9-month schedule. This requires two additional key steps. One is to continuously update the priorities of the features, so that no time is lost in deciding what to drop. The other is to organize the architecture to make it easy to drop the lower-priority features. Attempting to succeed at SAIV without having provided these necessary preconditions for successful corrective action has compromised the success of a number of entrepreneurial startups.

Design for Lifecycle Value

Developing a software system or portfolio of systems is an ongoing activity of design decision-making that extends across multiple organizational and product granularity levels and through time. The software economics viewpoint on this activity has two basic parts. Foremost is that the objective of software design is to create surplus value. The goal is not to achieve verifiability, evolvability, safety, quality, usability, reusability, reliability, satisfaction of a formal specification, possession of a mathematical semantics, or any other technical property, per se. Technical properties are of course critical to creating value, but they are the means, not the ends. The guiding objective for software engineering is *design for value added.*

The second part of the economics viewpoint tries to answer the question, How does one do design for value? One key answer is that one should treat software development as an *investment* activity, and look for real improvements by modeling, analyzing, and managing it as such. We take *active investment management* as a central part of an approach to achieving value creation objective. By active investment management we mean structuring a mix of products, processes, projects, and portfolios, and operational targets to enable ongoing creation and exploitation of favorable investment opportunities, of synergies among concurrent projects, and of synergies among sequential projects.

Much work has been done in economics—especially finance—to develop tools for reasoning about value creation through intelligent, dynamic investment decision-making. Relevant issues include how present and future costs and benefits are made commensurable; how risk and risk tolerance can be characterized, measured, managed and factored into valuations; how present value can be created in the form of exposure to opportunities for future gain and protection against down-side risks; how efficient capital structures that promote speed and variety in innovation can be fostered; and so on.

At the same time, the software engineering field has pushed our understanding of relationships between product and process structure and value in the face of complexity, uncertainty, and competition—albeit in an ad hoc and informal manner. Some of the seminal work in software engineering has been picked up by finance researchers. Recent research on economic drivers of the evolution of the computer industry, for example, point to information hiding modularity as embodied in open architectures as being fundamental [6]. It is thus natural for software engineering researchers to seek out analogies and correspondences between investment concepts and software design in an attempt to leverage knowledge from finance to improve productivity in the software domain.

1 Value-Driven Design

While the finance concepts of cost and benefit are well known, a sophisticated economic perspective raises issues that are not typically addressed adequately in software development projects today. Advances are possible when a more sophisticated economic viewpoint leads to the consideration of complex sources of value, and the means by which value creation can be improved. By exploiting modern finance concepts, software engineers can develop a better understanding of such issues as the following:

- the present value of future payoffs that might or might not be attained
- the value of new information that reduces uncertainty about unknown states of nature
- the value of risk reduction with all other factors, including expected payoffs, the same
- the present value of options whose payoffs depend on how exogenous uncertainties are resolved, including options to enter new markets if market conditions become favorable; to make follow-on investments if exploratory phases produce favorable results; to abandon money-losing investments for abandonment payoffs; to ship a product early or just to be able to make a credible threat of doing so
- how desired risk-return characteristics can be attained through diversified portfolios of assets, provided that the assets have somewhat independent returns
- the nonlinear value of networks in their size [70].
- the opportunity cost of investing early in the face of uncertainty, and of investing late in the face of possible drops in asset values, as might result from competitive entry into a market

If these concepts are to be exploited by software engineers, then it is important to relate them to terms and decision criteria that software engineers understand. Important software engineering decision-making heuristics include the following:

- information hiding modularity
- architecture first development
- incremental software development
- always having a working system
- risk-based spiral development models
- the value of delaying design decisions
- components and product-line architectures

Modularity and architecture, in particular, have strategic value in establishing valuable options: to improve a system one part at a time as new or revised parts are delivered; to create variants to address new or changed markets; and to develop variants quickly. Phased project structures, as promoted especially by the spiral development model and its variants, create valuable options in the form of intermediate decision points. It's far less costly to abandon a relationship before becoming engaged than after, before getting married than after, before having children than after. The value maximizing decision is to cut one's losses early as soon as it is clear that a project is not going to succeed. A culture that legitimizes abandonment as a potentially value-creating alternative is better than one that makes abandonment difficult by equating it with failure.

Relating software design concepts to value-based analogs opens up considerations that have significant potential to inform software development, especially in the strategic dimension. Rather than thinking of design as an anticipatory activity that succeeds if one anticipates correctly and that fails if not, for example, one can reason about the increased *present* value of a design that has flexibility to accommodate changes that might or might not occur. The designer succeeds if the value of the system is increased (net of cost and risk) by the decision to include flexibility that has a clear although uncertain potential to produce a future payoff.

2 *Investing in the Anticipation of Change*

Although the distinction is subtle, it is important. Among other things, it emphasizes the need for the designer to manage the ever-changing valuations of the elements of a software development portfolio to optimize for value. If an "anticipated" change does not occur, then the value of the system decreases because the flexibility no longer holds the potential for a future payoff. The value of the system decreases because of the change in external circumstances. When such a change is significant enough, the design situation in the small or even in the large might need to be reconsidered. The ongoing valuation of one's assets, the monitoring of conditions that affect those valuations, and adjustments in the asset mix and operational objectives are the keys to an active investment management approach to software design decision-making.

At the corporate level, this view puts a premium on investments that enable designers to better anticipate change. Examples are evaluation of emerging technologies (CORBA, COM, object management systems, agent management systems), and product usage trend analysis.

Of course, there is no silver bullet. Software design is and will continue to be an exceedingly demanding activity. Although financial concepts of value and management for value added can contribute to software design, they are no panacea. The complexity of the activity ensures that there is no simple formula. Many factors have to be brought together at once for software or software-enabled systems to deliver the benefits that, net of their costs, produce a surplus that is then distributed among the stakeholders to make everyone better off. Complexity ensures that there are many ways that things can go wrong to undercut the attainment of benefits. Ensuring that all of the required factors are aligned will remain a challenge of the first order.

5 EMERGING VALUE-DRIVEN DESIGN AND DEVELOPMENT APPROACHES

The second key issue with respect to optimizing for value, then, is to understand the entire set of conditions that must

occur together for benefits to be realized. Timely and economical production of a product having the properties needed to satisfy a need is of the essence, of course. What sets of conditions must be orchestrated for the successful delivery of a value-creating product? What failure modes threaten such an efforts? Design approaches now emerging from industry and from academic research laboratories are beginning to tackle these question head-on, with an emphasis on the value creation through a comprehensive approach to ensuring the realization of defined benefits.

Two such efforts are Model-Based (System) Architecting and Software Engineering (MBASE) [12] and the "benefits realization" approach of Thorp and DMR Consulting [79]. The focus of each is on achieving all of the conditions necessary for defined benefits to be realized. We discuss each approach in turn. We compare and contrast them with each other. Then we put them in the broader context of active investment management. We then discuss what some of the challenges might be in reorganizing existing enterprises to follow such approaches.

MBASE/USC

The Model Based (System) Architecting and Software Engineering (MBASE) approach [12] is driven by the view that a narrow focus on technical aspects of system architecture is far from enough to promote successful outcomes. Instead, the approach advocates a holistic treatment of all of the key issues, in four basic dimensions, that must be addressed for success to occur.

The *product* dimension addresses issues such as domain model, architecture, requirements, and code. The *process* dimension involves tasks, activities, and milestones. The *property* dimension involves cost, schedule, and performance properties. The *success* dimension addresses what each stakeholder needs, e.g., in terms of business cases, satisfaction of legal requirements, or in less formal terms. The basic idea is that in each of the dimensions, activities and expectations are guided by models, and that these models must be mutually consistent for a project to succeed.

The central axiom of the MBASE approach is that success depends on the contributions of multiple self-interested parties—stakeholders—and that for the required set of contributions to materialize, all stakeholders must be satisfied that the process will satisfy their success criteria. It becomes critical to understand the success models of each success-critical stakeholder and to manage the activity to sustain each stakeholder's expectation of success. Recognizing conflicts among success models, reconciling them, and managing expectations emerge as key challenges.

More generally, the approach recognizes that one of the key sources of problems in complex projects is in the misalignment or incompatibility of models. The approach thus emphasizes the need to bring the models into harmony so that they reinforce each other. Stakeholder success models along with application domain models drive choices of property, process and product models. The approach provides guidance for identifying and resolving conflicts, and heuristics for promoting the coherence of the multiple models. As clashes among success criteria are resolved, for example, the consistent set of criteria are embodied in product models, e.g., in the system specification.

The next important concept is that the elaboration of a harmonious set of models cannot occur either sequentially or in a single "big bang." Things change and people cannot foresee all issues that will arise in a complex development effort. The MBASE approach thus emphasizes the use of an approach in which models are elaborated and made ever more mutually reinforcing over time in an iterative fashion. The Win-Win spiral model is employed in an attempt to ensure that the ultimate objectives—each stakeholder's desire for the satisfaction of its success criteria—is continually accounted for in the evolving set of models. As conditions evolve, success criteria, requirements, processes, and other model elements are adjusted in an attempt to keep the effort on track.

The MBASE approach recognizes that conditions change and that early visions might or might not succeed. Thus it incorporates as a key element in the iterative development process a three major *anchor point* milestones: life cycle objectives (LCO), life cycle architecture (LCA), and initial operational capability (IOC). These milestones represent fundamental stakeholder life cycle commitment points analogous to the real-life commitment points of getting engaged, getting married, and having your first child. The Anchor Point milestones define constituent elements and associated reviews and pass/fail conditions. LCO and LCA include as essential content an Operational Concept Definition, Requirements Definition, Architecture Definition, Life Cycle Plan, Key Prototypes, and Feasibility Rationale, Architecture Review Board reviews and their Feasibility Rationale-based pass-fail criteria. The IOC milestone addresses the deliverables for software, personnel, and facility preparation, and it includes a Transition Readiness Review, a Release Readiness Review, and their associated pass-fail criteria.

The final MBASE core invariant is that the design and content of MBASE artifacts and activities should be driven by considerations of risk management. The rationale is that the risk criterion is the best way for a project to determine the adequacy of specifying, prototyping, reusing, testing, documenting, reviewing, and so on. The failure to apply this criterion increases the risk that a project will not achieve critical success conditions and the risk that effort will be wasted on unnecessary or dysfunctional activities.

DMR/Thorp

Thorp traces the evolution of the application of information technology from automation of routine work, through information management to its primary role today: enabling profound transformations in business structures and functions. His primary thesis is that the expected benefits of information technology generally are not being realized today because (1) realizing the benefits of information-technology-enabled business transformation can require coordinated change across an entire organization, not just the installation of new information technology components; and (2) managers continue to behave as if they were still in the old world of work automation or information management, when simply installing computers and software was perhaps adequate to realize benefits.

In particular, Thorp emphasizes the need for manager to consider four key issues. The first is the *linkages* between information technology investments and business strategy on one hand and, on the other, changes needed elsewhere in the organization (e.g., training) that are required to realize benefits. The second is *reach*: the extent to which an IT-enabled change impacts on the organization, both in the range of business units and functions affected and in the depth of changes required. Understanding and managing the impacts of such changes, e.g., on involved stakeholders, is seen to be critical. The third issue is *people*: that many people must commit to a given transformational change, and that can require significant engineering of attitudes, etc. The fourth issue is *time:* that the time frame within which a change is made is critical to success, but hard to predict in advance, and it is not infinitely flexible, as organizations can absorb change at a finite rate. Thorp also emphasizes that the parameters in these dimensions will themselves change over time.

The Thorp/DMR benefits realization approach is based on several premises: first, technology alone is insufficient to produce benefits; second, early visions of benefits are rarely realized, but rather the benefits that are ultimately achieved are based on a dynamic, somewhat unpredictable process of benefits pursuit over time; and third, realizing benefits requires a continuous process of envisioning the benefits desired, implementing, and dynamically adjusting course in light of new information.

The first point drives a change in perspective from traditional engineering, based on a cycle of design-develop-test-deliver, to an end-to-end view of technology-intensive development: *from concept to cash.* The approach is holistic and focused on realizing value, rather than just on technical issues. It recognizes that the strategic value of information technology is increasing exponentially, but that as its impacts cut deeper and ever more broadly across organizations, and as the costs of information technology continue to drop, the fraction of the cost of IT-enabled change attributable to IT itself continues to dwindle.

At an investment structuring level, the Thorp/DMR approach is based on several concepts. The first is that the emphasis has to shift from stand-alone *projects*, for which the goal is the delivery of a discrete capability (e.g., a software system), to multi-project *programs,* for which the goal is to produce benefits at the organizational level. A project might deliver a computer, but it takes a program to put a person on the moon.

The second idea is that an organization should take a disciplined approach to designing portfolios of programs in order to realize given benefits while managing risk and return. The programs in these portfolios, as suggested above, will combine investments in information technology with other initiatives as necessary to achieve defined business objectives.

The third core concept is that a *full cycle governance* approach be taken to managing the portfolio and its constituent programs. This idea combines active, full life cycle management with the notion of a phased and incremental investment approach based on well defined *stage gates.* Stage gates are major evaluation and decision points for programs that enable reevaluation of changes in the state of a program and its environment, and decisions about whether to change course, e.g., to abandon, redirect, or reinforce a program.

In terms of management, per se, the Thorp/DMR approach requires what he calls *activist accountability, relevant measurement,* and *proactive management of change.* Activist accountability means that a senior business manager owns each program and is accountable for the realization of its benefits. An important corollary is that the information technology group cannot reasonably be held accountable for realizing business benefits of IT-enabled change. It does not have the ability or authority and scope of control to coordinate all of the elements needed to realize benefits at the organizational level; so it must not be given such responsibility, either.

The measurement issue stresses that the tracking of performance parameters related not just to costs but to benefits is critical. Costs are tangible; performance measures that reflect the realization of benefits are harder to find.

The third issue is proactive management of change. This issue goes to the question of implementing a benefits realization approach or any of the major IT-enabled changes within it, in an organization not already set up for such changes. Senior leadership is seen to be essential is managing change across the dimensions of linkages, reach, people and time.

Synthesis

The MBASE and Thorp/DMR approaches overlap considerably in the concerns that they address. First and

foremost, both take a holistic view of design, acknowledging that many factors at once have to be addressed for design investments to pay off. They both take phased approaches to scaling up commitments only as they scale down risks. They achieve this outcome through phased investment structures with defined milestones. The DMR approach emphasizes strategy, risk engineering, and the creation and management of synergies and real options more strongly than MBASE. On the other hand, MBASE is significantly stronger in addressing particular issues that arise in the context of complex software components of software-intensive system projects.

6 RESEARCH CHALLENGES

A broad set of research challenges is arising in software economics. First, software economics has always addressed important problems, but the major economic discontinuities created by software and information technology itself are now pushing software economics to the top of the research agenda. Second, the problems are of real scientific and technological—not just economic—interest. Design activities are in some sense guided by "force fields" [6]. The proper force field for guiding software engineering, as for any utilitarian design and engineering discipline, is value creation, measured in terms that count for the enterprise that is investing resources. A science of design, which is, after all, what we seek as a basis for software engineering, will have to account for the influence of that field. Third, the practical significance of advances, or of the failure to make them, is high. Fourth, it appears possible to make significant progress through a comprehensive program in research and education in software economics.

A research program will include several elements. At the highest level, we need to learn how to think about and manage software development as an *investment activity,* the goal of which is to create maximum value for the resources invested. Reasoning about how to manage investment to maximize value is becoming very sophisticated, and many kinds of enterprises, including philanthropic foundations, are borrowing from the advances that are being made. As Brooks observed in regard to software, it is easier to buy than to build. Perhaps that is also true for important concepts and structures for organizing complex investment activities. In other words, there might be much to gained by borrowing from other fields and by adapting existing knowledge to the software context.

Software development is a complex activity, so it should not be astonishing to find that significant adaptation is necessary in some cases. Other fields, such as finance and strategy, can help to inform software design. In return, software design and engineering can inform other fields. Baldwin and Clarke, in developing a theory of the evolution of the computer industry, in particular, appeal directly to the information hiding notion.

In particular, the role of strategy in value creation is now much better understood and appreciated than it was thirty years ago. Strategy involves coordinated and dynamically managed investments in multiple, interrelated areas with specific value creation objectives. It is now understood that much of the present value of some enterprises is in the form of options that they hold to profit from possible future opportunities [51], and that a fundamental strategy for increasing value is to invest in the creation of such options and in the capabilities necessary to exercise them. Capabilities extend all the way to culture. An enterprise can do better than its competitors, for example, if its culture accepts project abandonment in light of unfavorable new information as a positive-value action.

Beyond the management activities of any individual enterprise, there is the larger question of how best to improve the design space in which they operate. Traditional software engineering research plays a vital role in enabling new and better technologies and processes; but innovative research on what is necessary to achieve a transformation of the industry structure is also needed. Understanding how to create more liquid markets in software risk is one potentially important step.

Within the enterprise, there are interesting questions about how best to allocate scarce resources at the "micro" level. Many activities contribute to the development of a software product. They all compete for resources. The value added is a complex function of the distribution of resources over those activities. Getting a sense of what that function is, both overall, and for particular life-cycle activity, such as verification, is important. Of course, the function will vary from one context to another, just as cost functions do.

To support all of these activities, new models and associated analysis methods are needed for reasoning about the value of investments in software technical assets, including design aspects of software processes, products, projects, programs and portfolios. Supporting tools and environments will then be needed to make the models useful to engineers in practice.

Finally, understanding how to integrate such advances into industrial software development processes is essential to realizing the benefits enabled by such research. It is not enough to produce disembodied models. Recent advances represented in the MBASE and DMR approaches provide indications that we are now at a stage where we can envision achieving such an integration.

7 EDUCATION

Finally, we note, briefly, that the research cannot have the desired impact without a coordinated investment in education. At a minimum, a traditional course in engineering economics would help to give software engineers a rudimentary background in finance. Other engineering disciplines consider engineering economics to

be an essential part of their respective bodies of knowledge. Recent attempts to define bodies of knowledge and related curricula for software engineering, by contrast, under-emphasize software engineering economics. For example, it might be treated within a course on software engineering management, typically in the form of attention to cost, schedule and risk estimation and management.

Such treatments would further delay the benefits to be gained from a holistic treatment of economics throughout the body of knowledge. Rather than presenting information hiding as an independent concept (it makes software easier to change in ways that you anticipate), it could also be justified in economic terms. Not every concept in software engineering succumbs directly to an economic analysis, of course. Giving students a visceral understanding of how to design for value in the face of uncertainty, incomplete knowledge and competition through clever product, process and portfolio design is a significant challenge that is not likely to be met by a curriculum that separates economic thinking from design and other aspects of development, and that leaves the treatment of economics with traditional cost and schedule estimation models and risk management concepts.

8 CONCLUSIONS

The software field exists because processed information has value. If people were not willing to pay for software development in the expectation of enhanced value, all of us in the software field would be out of jobs.

Given that we live in and benefit from this value-determined situation, it is in our enlightened self-interest to increase our understanding of and our ability to deal with the economic aspects of software, its development, and use.

The results chain of initiatives, contributions, and outcomes in Figure 1 is indeed a roadmap for how we can progress from our current barely-coping stage of software economics mastery to a world in which more informed software and information technology decisions lead to much greater value creation and quality of life for all of us.

However, as with other results chains, we must be careful to ensure that the assumptions underlying the achievement of the outcomes are valid. The biggest assumption underlying the roadmap in Figure 1 is *that there are enough people with a sufficient understanding of both software and economic phenomena to enable the contributions and outcomes to be realized.*

As we have discussed, one step in this direction is to enable more software people to emerge from an economics-unaware logical Flatland and better deal with the economic aspects of software. But this is not enough. There are also many people living in other Flatlands, in which they may do well at marketing or finance, but have an insufficient understanding of software phenomenology to function well at creating value via software.

By improving and propagating our understanding of software phenomenology and its economic aspects, we can evolve to where we can all live in a fully-dimensional world spanning software and economic phenomenology, and advance our abilities to generate value via information technology many-fold.

ACKNOWLEDGEMENTS

We acknowledge the contributions of the participants and organizers of the First Workshop on Economics-Driven Software Engineering Research for valuable discussions and for suggesting some of the important issues that we discuss in this paper. This work was supported in part by the National Science Foundation under grant CCR-9804078. Mary Shaw emphasized the need to consider the issue of markets in risk for software. Somesh Jha has discussed with us his investigations into a portfolio analysis approach to allocating resources to activities in software projects. Elizabeth Teisberg has provided valuable feedback to Sullivan on the topic of real options. Boehm's contributions have been supported by the Defense Advanced Research Projects Agency, the Federal Aviation Administration, the National Science Foundation, the Office of Naval Research, and the Affiliates of the USC Center for Software Engineering. The U.S. Government is authorized to reproduce and distribute reprints for Governmental purpose notwithstanding any copyright annotation thereon. The views and conclusions contained herein are those of the authors and should not be interpreted as necessarily representing the official policies or endorsements, either express or implied, of the Defense Advanced Research Projects Agency or the U.S. Government.

REFERENCES

1. T.Abdel-Hamid and S. Madnick, Software Project Dynamics, Prentice Hall, 1991.
2. M. Amram and N. Kalutilaka, Real Options, Harvard Business School Press, Cambridge, Mass., 1999.
3. K.J. Arrow, "Economic Welfare and the Allocation of Resources for Invention," in The Rate and Direction of Inventive Activity: Economic and Social Factors, NBER, Princeton University Press, 1962, pp. 609-626.
4. R. Austin, Measuring and Managing Performance in Organizations, Dorset House, 1996.
5. R. Axelrod, The Evolution of Cooperation, Basic Books, 1985.
6. Baldwin, C. and K. Clark, Design Rules: The Power of Modularity, MIT Press, 1999.
7. V. Basili, C. Caldeira, and H. D. Rombach, "Goal Question Metric Paradigm," in J. Marciniak (ed.),

Encyclopedia of Software Engineering, John Wiley and Sons, 1994, pp. 528-532.

8. B.W. Boehm, Software Engineering Economics, (Upper Saddle River, New Jersey: Prentice Hall PTR), 1981.
9. B.W. Boehm. "A spiral model of software development and enhancement." IEEE Computer, pages 61-72, May 1988.
10. Boehm, B., Egyed, A., Kwan, J., Port, D., Shah, A., and Madachy, R., "Using the WinWin Spiral Model: A Case Study," IEEE Computer, July 1998, pp. 33-44.
11. B. Boehm and R. Ross, "Theory-W software project management: principles and examples," IEEE Transactions on Software Engineering, 15(7):902-916, July 1989.
12. Boehm, B. and Port D., "Escaping the Software Tar Pit: Model Clashes and How to Avoid Them," Software Engineering Notes, Association for Computing Machinery, pp. 36-48, January 1999. See also http://sunset.usc.edu/MBASE.
13. B. Boehm and K. Sullivan, "Software Economics: Status and Prospects," Information and Software Technology 41, 1999, pp. 937–946.
14. B.W. Boehm, C. Abts, A.W. Brown, S. Chulani, B. Clark, E. Horowitz, R. Madachy, D. Reifer, and B. Steece, Software Cost Estimation with COCOMO II, Prentice Hall, 2000.
15. R.A. Brealey and S.C. Myers, Principles of Corporate Finance, 5th edition, McGraw Hill, 1996.
16. "Software Hell," Business Week, December 6, 1999, pp. 104-118.
17. E. Carmel, R. Whitaker, and J. George, "PD and Joint Application Design: A Transatlantic Comparison," Communications of the ACM, June 1993, pp. 40-48.
18. R. Chillarege, I. Bhandari, J. Chaar, M. Halliday, D. Moebus, B. Ray, and M. Wong, "Orthogonal Defect Classification—A Concept for In-Process Measurements," IEEE Transactions on Software Engineering, November 1992.
19. S. Chulani, B. Boehm, and B. Steece, "Calibrating Software Cost Models Using Bayesian Analysis," IEEE Transactions on Software Engineering., July-August 1999, pp. 573-583.
20. J.C. Collins and J.I. Porras, Built to Last: Successful Habits of Visionary Companies, Harper Business, 1997.
21. W. R. Collins, K.W. Miller, B.J. Spielman and P. Wherry, "How good is good enough?: An ethical analysis of software construction and use," Communications of the ACM, vol. 37, no. 1, January, 1994, pp. 81 – 91.
22. R. Cruickshank and J. Gaffney, "An Economics Model of Software Reuse," in T. Gulledge and W. Hutzler (ed.), Analytical Methods in Software Engineering Economics, Springer-Verlag, 1993, pp. 99-137.
23. M.A. Cusumano and RW. Selby, Microsoft Secrets, (New York, New York: The Free Press), 1995.
24. A.K. Dixit and R.S. Pindyck, Investment Under Uncertainty, (Princeton, New Jersey: Princeton University Press), 1994.
25. D.F. D'Souza and A.C. Wills, Objects, Components and Frameworks with UML: The Catalysis Approach, Addison Wesley, Reading, Mass., 1999.
26. A.F. Egyed and B.W. Boehm, "Telecooperation experience with the win-win system," Proceedings of the IFIP World Computer Conference, September 1998.
27. W. Eureka and N. Ryan, The Customer-Driven Company: Managerial Perspectives on QFD, ASI Press, 1988.
28. P. Ferguson, G. Leman, P. Perini, S. Renner, and G. Seshagiri, "Software Process Improvement Works!" CMU/SEI-99-TR-026, November 1999.
29. F.R. Freiman and R.E. Park, "PRICE Software Model-Version 3: An Overview," Proceedings, IEEE/PINY Workshop on Quantitative Software Models, IEEE Catalog No. TH0067-9, October 1979, pp. 32-44.
30. David Garlan, Robert Allen, and John Ockerbloom. "Architectural mismatch: Why reuse is so hard," IEEE Software, 12(6):17-26, November 1995.
31. C.C.Gotlieb, The Economics of Computers: Costs, Benefits, Policies, and Strategies, Prentice Hall, 1985.
32. R. Grady, Practical Software Metrics for Project Management and Process Improvement, Prentice Hall, 1992.
33. Y.Y. Haimes, Risk Modeling, Assessment, and Management, Wiley, 1998.
34. J.S. Hammond, R.L. Keeney and H. Raiffa, Smart Choices: A Practical Guide to Making Better Decisions," Harvard Business School Press, 1999.
35. C.J. Hitch and R.N. McKean, The Economics of Defense in the Nuclear Age, Harvard University Press, 1960.
36. I. Jacobson, M.L. Griss, and P. Jonsson, Software Reuse, Addison Wesley, 1997.

37. R.W. Jensen, "An Improved Macrolevel Software Development Resource Estimation Model," Proceedings, ISPA 1983, April, 1983, pp.88-92.

38. C. Jones, Programming Productivity, McGraw Hill, 1986.

39. R.L. Keeney and H. Raiffa, Decisions with Multiple Objectives: Preferences and Value Tradeoffs, (Cambridge England: Cambridge University Press), 1993.

40. J.P.C. Kleijnen, Computers and Profits: Quantifying Financial Benefits of Information, Addison Wesley, 1980.

41. J.C. Knight, K.J. Sullivan, M. Elder, "Survivability Architectures," Proceedings of DARPA Information Survivability Conference and Exposition (DISCEX), January 25—27, 2000.

42. B.P. Lientz and E.B. Swanson, Software Maintenance Management, Addison-Wesley, Reading, Mass., 1980.

43. W.C. Lim, Managing Software Reuse, Prentice Hall, 1998.

44. R. Madachy, "System Dynamics Modeling of an Inspection Process," Proceedings, ICSE 18, March 1996, pp. 376-386. [Malan-Wentzel, 1993]. R. Malan and K. Wentzel, "Economics of Software Reuse Revisited," Proceedings, 3rd Irvine Software Symposium, UC Irvine, April 1993, pp. 109-121.

45. F. Machlup, The Production and Distribution of Knowledge, Princeton University Press, 1962.

46. R. Malan and K. Wentzel, "Economics of Software Reuse Revisited," Proceedings, 3rd Irvine Software Symposium, UC Irvine, April 1993, pp. 109-121.

47. J. Marschak, Economic Information, Decision, and Prediction, 3 vol. (1974).

48. J. Marschak and R. Radner, Economic Theory of Teams, Yale University Press, 1972.

49. F. McGarry, R. Pajerski, G. Page, S. Waligora, V. Basili, and M. Zelkowitz, "Software Process Improvement in the NASA Software Engineering Laboratory," CMU/SEI-94-TR-22, December 1994.

50. J. Musa, A. Iannino, and K. Okumoto, Software Reliability, McGraw Hill, 1987.

51. Myers, S., 1977, "Determinants of corporate borrowing", Journal of Financial Economics, 5, pp. 147–75.

52. E.A. Nelson, Management Handbook for the Estimation of Computer Programming Costs, AD A-648750, Systems Development Corp., October 31, 1966.

53. D.L. Parnas, "On the criteria to be used in decomposing systems into modules", Communications of the Association of Computing Machinery, 15(12) pp. 1053–58.

54. D.L. Parnas, "Designing Software for Ease of Extension and Contraction," IEEE Transactions on Software Engineering, March 1979, pp.128-137.

55. M. Paulk, C. Weber, B. Curtis and M.B. Chrissis, *The Capability Maturity Model: Guidelines for Improving the Software Process,* Addison-Wesley, June 1995.

56. M. Phister, Jr., Data Processing Technology and Economics, Digital Press, 1979.

57. President's Information Technology Advisory Committee, Report to the President, Information Technology Research: Investing in Our Future, February 24, 1999.

58. J.S. Poulin, Measuring Software Reuse: Principles, Practices and Economic Models, (Reading, Massachusetts: Addison Wesley), 1997.

59. M.E. Porter and M.R. Kramer, "Philanthropy's new agenda: creating value," Harvard Business Review, November-December, 1999, pp. 121–130.

60. L.H. Putnam, "A General Empirical Solution to the Macro Software Sizing and Estimating Problem, IEEE Transactions on Software Engineering, July 1978, pp. 345-361.

61. Howard Raiffa. Decision Analysis: Introductory Lectures on Choices Under Uncertainty, McGraw-Hill, Inc.: New York, 1968.

62. J. Rawls, A Theory of Justice, Belknap, 1999 (revised edition).

63. D.J. Reifer, Practical Software Reuse, John Wiley and Sons, 1997.

64. R.W. Rimel, "Strategic Philanthropy: Pew's approach to matching needs with resources," Health Affairs, vol. 18, no. 3, May-June, 1999, pp. 228–233.

65. W. Royce, Software Project Management: A Unified Framework, (Reading, Massachusetts: Addison-Wesley), 1998.

66. H.A. Rubin, "A Comparison of Cost Estimation Tools," Proceedings, ICSE 8, August 1985, pp.174-180.

67. Software Engineering Institute, "Defining and Using Software Measures," CMU/SEI-92-TR, -11, -19, -20, -21, -21, -22, -23, -25, 1992.

68. Software Engineering Institute, CMM for Software, Version 1.1, SEI-93-TR-24 and –25, 1993.

69. Software Engineering Institute, Capability Maturity Model—Integrated—Systems/Software Engineering, Version 0.2b, September, 1999 (http://www.sei.cmu.edu/cmm/cmmi).

70. C. Shapiro and H.R. Varian, Information Rules: A Strategic Guide to the Network Economy, (Cambridge, Massachusetts: Harvard University Press), 1999.

71. W.F. Sharpe, The Economics of Computers, Columbia University Press, 1969.

72. M. Shaw and D. Garlan, Software Architecture, Prentice Hall,1996.

73. H.A.Simon, The Sciences of the Artificial, MIT Press, 1988.

74. G.J. Stigler, "The Economics of Information," Journal of Political Economy, vol.69, pp. 213-225.

75. P.A. Strassman, The Squandered Computer, (New Canaan, Connecticut: The Information Economics Press), 1997.

76. K.J. Sullivan, P. Chalasani, S. Jha and V. Sazawal, "Software Design as an Investment Activity: A Real Options Perspective," Real Options and Business Strategy: Applications to Decision Making, L. Trigeorgis, ed., (London, England: Risk Books), 1999, pp. 215—261.

77. K.J. Sullivan, J.C. Knight, X. Du and S. Geist, "Information Survivability Control Systems," Proceedings of the 21st International Conference on Software Engineering, pp. 184--193, May 1999.

78. E.O. Teisberg, "Methods for evaluating capital investment decisions under uncertainty," in Real Options in Capital Investment: Models, Strategies, and Applications, L. Trigeorgis, ed., (Westport, Connecticut: Praeger), 1995.

79. J. Thorp and DMR's Center for Strategic Leadership, The Information Paradox: Realizing the Benefits of Information Technology, McGraw-Hill, 1998.

80. Real Options in Capital Investment: Models, Strategies, and Applications, L. Trigeorgis, ed., (Westport, Connecticut: Praeger), 1995.

81. L. Trigeorgis, Real Options: Managerial Flexibility and Strategy in Resource Allocation, (Cambridge, Massachusetts: MIT Press), 1997.

82. G. Weinberg and E. Schulman, "Goals and Performance in Computer Programming," Human Factors, 1974 (16) 1, pp. 70-77.