Software economics: status and prospects

B. Boehm*, K. Sullivan

Center for Software Engineering, University of Southern California, Los Angeles, CA 90089-0781 USA

Keywords: Software economics; Decision making; Cost estimates

1. Overview

Software is valuable when it produces information in a manner that enables people and systems to meet their objectives more effectively. Software engineering techniques have value when they enable software developers to build more valuable software. Software economics is the sub-field of software engineering that seeks improvements which enable software engineers to reason more effectively about important economic aspects of software development, including cost, benefit, risk, opportunity, uncertainty, incomplete knowledge and the value of additional information, implications of competition, and so forth. In this paper, we survey the current status of selected parts of software economics, highlighting the gaps both between practice and theory and between our current understanding and what is needed.

The sheer volume of current software costs makes the study and application of software economics techniques a significant area of concern and opportunity. Recent studies [1,2] estimate roughly 2,000,000 software professionals in the US in 1998. At typical salaries of $60–80,000/year and a typical overhead rate of 150%, this translates into a $300–400 billion annual expenditure on software development in the US alone. A conservative estimate of worldwide software costs is twice the US costs, or $600–800 billion per year.

With the kind of expenditures now being made on software, just the economics of more efficient software production are important to understand and apply. Software development is widely seen to be inefficient, which means that there is considerable room for improvement. A 10% reduction in software production costs translates into a $30–40 billion annual savings in the US, and $60–80 billion worldwide.

One cause of inefficiency is the inadequacy of economic information available to support decision making. In the absence of reasonable cost estimates, for example, projects are at risk. Improving our basic understanding of software economics and the abilities of organizations to model and analyze economic aspects, can help to make them significantly more productive. A simple example: once organizations appreciate that late defect fixes are more expensive than earlier fixes, they can reduce their costs by reorganizing their process to focus on early defect prevention and elimination.

The software engineering activity centers on decision making in very complex situations. What requirements are feasible given certain resource limitations? How much should one bid on a project? How should one modularize a design to create opportunities and to reduce future costs? How can processes be structured to dramatically improve cycle times so as to capture profits in ephemeral markets?

Such decisions can be addressed more effectively using economics-based techniques; one definition of “economics” is “the study of how people make decisions in resource-limited situations.” An important problem is to empower high-level managers to choose the best available economic reasoning techniques for use in their projects. This paper is meant to help researchers and managers by providing an overview of the current state and prospects for software economics. We address three main questions. In Section 2 we discuss how we have done up to now in software economics, and what forces are driving the need for new work. Section 3 focuses on the current state of our knowledge and practice in cost and schedule estimation, in particular, and in decision support, more broadly. Section 4 closes with a more general perspective of future trends.

2. Changing landscape

Increasing competition is driving organizations to focus intensely on the connections between technical activities and economic results. Past research on, and related to, software economics has produced significant advances in knowledge that have enabled organizations to produce software more effectively. However, changes occurring in technology and business are widening the gaps between...
traditional software economic models and contemporary software development. In this section, we summarize the strengths of past work that has been done in software economics. We highlight changes demanding new attention to software economics. We discuss particular weaknesses in software economics needing attention. Finally, we comment on the increasingly clear need for long-term, basic research in software economics.

2.1. Existing strengths

As we discuss in Section 3, the economics of software production using cost estimation models is one of the more mature areas of software economics. Several models have proven to be fairly reliable and valuable in traditional development projects. The concept of risk-driven development processes, such as the spiral model, is now well accepted in theory and practice. Traditional microeconomics and operations research concepts have been applied in software engineering. Business data processing has long taken net present value as a measure of the worth of information technology projects; and the concept of decision making under uncertainty with experimentation provides a model of the value of investments in risk reducing prototypes. Negotiation-based approaches to requirements and other aspects of software development reflect our understanding that preferences vary among stakeholders, and that project success often hinges on satisfying all of them.

2.2. Emerging conditions

Although cost estimation and related models have been successful, there are other software economics issues that are at least as important, and generally less well understood. Many of these issues are being driven by the changes that information technology itself is causing. Others are due to major political changes, such as the end of the Cold War, globalization, and the rise of the commercial sector as the dominant force in the world economy. Yet others are being driven by advances in our understanding of software engineering, including in particular the use of open-ended, iterative approaches to development, in contrast to the bounded, waterfall structures of the past. One consequence is that cost estimation models that were successful in traditional development projects are increasingly challenged by new development processes. Even more seriously, cost is no longer the high-order risk to the success of many projects.

- In the world of Internet time, reducing software development schedules may be considerably more important to competitive success than reducing software cost. A number of Rapid Application Development strategies are emerging [3,4], for which traditional schedule estimation models are inadequate.
- Factors such as software delivery schedule, product features, and product quality are likely to have a much larger effect on an organization’s bottom-line than are software costs. The ability to reason about software-induced benefit flows and overall software returns on investment becomes paramount, but is considerably lagging the ability to estimate costs.
- The evolution toward an increasingly electronic marketplace not only makes software considerations more central, but also changes a number of fundamental assumptions about how products and services are marketed, produced, delivered, and paid for. New economic models are needed to reason about these new forms of cost and benefit flows.
- Factors such as COTS and outsourcing are increasingly attractive as options for accelerating software production in and for the electronic marketplace. However, they have critical controllability, compatibility, and predictability challenges whose economic aspects are important to address, but often quite slippery to formulate.
- The increasing level of uncertainty in the competitive marketplace and in the cost/value structure of purchased software places a high premium on the ability to reason about software uncertainty and risk. Some classical economic models apply to some extent, but more sophisticated models are needed.

2.3. Growing weaknesses

The difficulties cited above point to gaps in our ability to reason about software economics as comprehensively as is necessary. At the highest level, the focus on cost has tended to overshadow the importance of a fundamental concern with the value added by investments in software. Value added measures what is left when one accounts not only for all of the costs incurred by a given investment in software, but all of the benefits, their distribution over future times, their exposure to risks, the preferences of those who judge what things are worth, and the possibility of investing in other, mutually exclusive, opportunities.

In particular, it is critical for software engineers and managers to recognize that the value added by a major investment depends not only on technical success, but on market conditions and the industry structure when the project is completed. Moreover, it is critical for managers to understand the present value of a project that will not be completed until a future date, because, at least in publicly traded firms, a primary goal of the management is to increase the present value of the firm. The valuation of an ongoing project must account for all of these areas, factoring in all of the major uncertainties—risks and opportunities—to which the project is exposed. We lack adequate frameworks in which the issues are integrated systematically to support decision makers. We highlight a few specific areas in which shortcomings are clear.

These can be categorized into strategic decisions, spanning a number of projects and products, and tactical decisions, addressing issues arising at an individual project level.
Some examples of strategic decision issues are:

- How far should I stretch my current software architecture to try to capture new markets? (amazon.com is a good example).
- How broad a domain should I encompass with a prospective new product line architecture and set of reusable components?
- What technical choices should I make for my internal software enterprise architecture: CORBA/DCOM; object management systems; collaboration frameworks; client–server middleware?
- How should I evolve my portfolio of software productivity investments: process maturity; tools; componentry; software knowledge bases?
- Which market sectors should I be trying to enter or leave?

Some examples of tactical decision are:

- I understand how to design for correctness, e.g. using formal methods; and I know how to design for change, e.g. using information hiding modular architecture; but how do I design for maximum value in a highly competitive, rapidly evolving marketplace?
- I know that there are risks in depending on external components, and that they reduce the value of my product; but what are the upside opportunities, e.g. as presented by the possibility of rapid upgrades when new versions appear; and how do I value them?
- Once I have an efficient system in which it is no longer possible to gain in quality, function, and time to market simultaneously, what tradeoff maximizes value added?
- How much should I spend on periodic internal releases of a fully functional system to hedge against preemptive entry into the market by a competitor?
- How will the market value a given function, such as end-user programmability, and how can we design our system so as to find out the answer as quickly as possible?
- Connections between technical software engineering activities and the value added to the end user are often unclear. What is it worth to fix a given bug, for example?

2.4. Need for renewal

Software engineering researchers and practitioners are just beginning to appreciate the need to provide more advanced capabilities to reason about major software decisions in economic terms. A recent landmark “First Workshop on Economics-Driven Software Engineering Research” (EDSER-1) identified a number of increasing sources of both demand and supply for software economics analysis techniques [5]. However, even for the existing capabilities, there is still a significant lag between the state-of-the-art in software economics and its use in practice. The need for significant progress is increasingly clear. Economic objectives and constraints now dominate almost all software development. Even when national security is the primary goal, for-profit contractors must realize the national security capabilities, subject to such economic forces as decreasing defense budgets. Advances in software economics promise to aid developers in all sectors.

Even in areas of traditional strength in software economics, work remains to be done. One problem already mentioned is that the new software development techniques demand new estimation methods. Moreover, by implication, continually evolving production methods demand continually evolving estimating methods. In another dimension, most of today’s successful estimating procedures are largely empirical in nature. It would be good to have theoretical models that would capture, explain and permit us to reason from an understanding of the underlying dynamics, including for example noise and stability.

Moreover, there appear to be opportunities to put other aspects of software engineering that are not currently formulated in terms of economics, value, or utility on a sounder footing. Most software design principles provide guidance that is implicitly economic in its nature, but that guidance is almost never explicitly economic in formulation. The scientific foundations of these principles thus remain weak, making them hard to evaluate and thus hard to refine, qualify, or reject. It also makes it hard to teach many important ideas in the field with authority, because they cannot be presented as scientific knowledge. Rather, they are often in the form of highly questionable rules of thumb. By making the underlying economics explicit, we thus might not only gain deeper insights into software design, but also empower software designers to do better.

The rest of this paper addresses two questions. Where are we now in terms of two of the more familiar software economics areas: estimation and decision support; and what are the emerging trends and future research directions?
3. Status of selected software economics areas

Space limitations preclude detailed summaries of all the major areas of software economics. Here, we concentrate on two of the most significant areas: software cost and schedule estimation and software decision support.

3.1. Software cost and schedule estimation

Fig. 1 summarizes the major software cost and schedule estimation techniques currently used or being researched: expertise-based, model-based, regression-based, composite-Bayesian, learning-oriented, and dynamics-based.

3.1.1. Expertise-based methods

By far the most commonly used are the expertise-based techniques. These frequently are estimates based on analogy to the estimator’s experience on similar previous projects. A more detailed estimate may be made via a set of work breakdown structure (WBS) elements. These elements often have top-level cost estimating relationships, such as 3–5% of total cost going to configuration management; productivity rates per new line of code or function point; or the amount of test effort per test case.

The main strength of these techniques is that they are generally based on real experience, and that human judgement is often good at adjusting for special situations. Their main weaknesses are estimator bias and the representativeness of the experience upon which the estimates are based.

Some group-consensus techniques such as Delphi [6,7] are available to address sources of estimation error such as estimator bias and incompleteness. Having a group of experts independently estimate, discuss the nature of their estimate differences, and then re-estimate, will often compensate for estimator bias and incompleteness. However such methods can be highly inefficient if a large number of changes must be accommodated across time.

3.1.2. Model-based methods

There are a number of parametric models for software cost and schedule estimation which have passed a market test as viable commercial products. These include Checkpoint [1], Estimacs [8], PRICES [9], SEER, SLIM [10], Softcost [28], and various commercial versions of the original COCOMO [6] and current COCOMO II [11] model, such as COSTAR [25], Cost Xpert [26], and Estimate Professional [27].

Most of the models, except for COCOMO and COCOMO II, have proprietary internals. Most have been evolving for some time, and have grown increasingly similar with respect to their external cost driver parameters. Most have choices of sizing metrics covering source instructions, function points, and object-oriented size metrics; choices of program attributes such as domain, complexity, language, reuse, and required reliability; choices of computer attributes such as time and storage constraints and platform volatility; choices of personnel attributes such as capability, continuity, and experience; and choices of project attributes such as tools and techniques, requirements volatility, schedule constraints, process maturity, team cohesion, and multisite development. It has been interesting to note a “parameter creep” in many of the models as they have evolved.

Most models have data sets to which they have been calibrated. They tend to be more accurate on the data set to which they are calibrated than they were on independently assembled data sets (e.g. Ref. [12]). This is due both to calibration advantages and to difficulties in reconciling model definitions and assumptions (process endpoints, counting rules, labor categories included, accounting for uncompensated overtime, etc.).

Most of the models were originally formulated around waterfall (single-shot, build-to-requirements) model assumptions, and are increasingly challenged by new process models (incremental, evolutionary, spiral); applications generators (graphic user interface builders); commercial-off-the-shelf (COTS) product integration; legacy code adaptation; integrated product teams; and optimization around various combinations of cost, schedule, and quality. Some models are undergoing extensive reinvention (e.g. COCOMO II); others are adapting or generating new variant models.

3.1.3. Regression-based methods

Another attractive approach is to choose a functional form for a model which can be linearized (e.g. via logarithms for multiplicative and exponential models), and to apply linear regression to determine the model coefficients providing the best fit to one’s available data.

However, it can be difficult to assemble software project data and formulate associated models which satisfy the following set of experience-based critical success factors for regression-based models:

(i) A lot of data is available. This indicates that there are many degrees of freedom available and the number of observations is many more than the number of variables to be predicted. Collecting accurate data has been one of the biggest challenges in this field due to lack of funding by higher management, co-existence of several development processes, lack of proper interpretation of the process, etc.

(ii) No data items are missing. Data with missing information is often reported when there is limited time and budget for the data collection activity; or due to lack of understanding of the data being reported.

(iii) There are no outliers. Extreme cases are very often reported in software engineering data due to misunderstandings or lack of precision in the data collection process, due to different “development” processes, or due to exceptionally high or low complexity, personnel capability, etc.
(iv) The predictor variables are not correlated. Most of the existing software estimation models have parameters that are correlated to each other. This causes instabilities in the regression equations.

The most effective ways to strengthen regression results are to apply robust regression techniques (e.g., via elimination of outliers or reducing their effect by minimizing median rather than mean values); to invest further in ensuring clean and comparable data; and to reduce the number of variables to be predicted.

The latter technique may work well for a highly homogeneous software organization, but it causes a major difficulty for general estimation models. For example, if one eliminates a variable such as personnel continuity from one’s model, inevitably someone will ask a question such as, “I’m considering the use of special bonuses to increase personnel continuity on my critical projects. How much difference will this make according to this model?”

By eliminating the variable, the only answer is, “It doesn’t make any difference.” This is often a difficult position to defend. The need to address such questions is the main source of the “parameter creep” in parametric models discussed above.

3.1.4. Composite-Bayesian methods
An attractive alternative to pure regression methods is a Bayesian approach which combines the strengths of expertise-based methods and regression-based methods. The Bayesian approach provides a formal process by which an a priori expert-judgement can be combined with sampling information (data) to produce a robust a posterior model.

Bayesian analysis [13] is a mode of inductive reasoning that has been used in many scientific disciplines. A distinctive feature of the Bayesian approach is that it permits the investigator to use both sample (data) and prior (expert-judgement) information in a logically consistent manner in making inferences. This is done by using Bayes’ theorem to produce a ‘post-data’ or posterior distribution for the model parameters. Using Bayes’ theorem, prior (or initial) values are transformed to post-data views. This transformation can be viewed as a learning process. The posterior distribution is determined by the variances of the prior and sample information. If the variance of the prior information is smaller than the variance of the sampling information, then a higher weight is assigned to the prior information. On the other hand, if the variance of the sample information is smaller than the variance of the prior information, then a higher weight is assigned to the sample information, causing the posterior estimate to be closer to the sample information.

The Bayesian approach has been used in the most recent calibrations of the COCOMO II cost estimation model. The to a priori means and variances for the COCOMO II parameters were determined via a Delphi process involving a number of cost estimation experts. A test was performed comparing the 1997 COCOMO II regression model as calibrated to 83 data points, and a Bayesian model in which the regression data was used to produce posterior model from the prior Delphi-based model. For the 83 data points in the 1997 sample, the regression model (necessarily) performed better than the Bayesian model. However, when tested against the 161 data points in the 1999 COCOMO II data base, the Bayesian model performed significantly better [14].

Bayesian analysis can thus combine the advantages of “Standard” regression with prior knowledge of experts. It attempts to reduce the risks associated with incomplete data generating. Software engineering data is usually scarce and incomplete, and estimators are faced with the challenge of making good decisions using this data. Classical statistical techniques described earlier derive conclusions based on the available data. But, to make the best decision it is also advantageous to incorporate non-sample or prior information that is relevant.

Usually, a lot of good expert judgment-based information on software processes and the impact of several parameters on effort, cost, schedule, quality, etc. is available. This information does not necessarily get derived from statistical investigation and hence classical statistical techniques such as OLS do not incorporate it into the decision making process. Bayesian techniques make best use of relevant prior information along with collected sample data in the decision making process to develop a stronger model.

3.1.5. Learning-oriented methods
One learning-oriented method is case-based reasoning, in which one adaptively learns which cases among a sample of projects best fit the characteristics of one’s application domain (e.g., electronic commerce, industrial process control, traffic control). One can then use the cost or productivity characteristics of this subset of projects to estimate the cost or productivity of new projects in the domain.

Neural network techniques provide another emerging approach for software cost estimation. According to Gray and McDonell [15], neural networks provide the most common software estimation model-building technique used as an alternative to regression techniques. Neural network estimation models can be “trained” using historical data to produce ever better results by automatically adjusting their algorithmic parameter values to reduce the delta between known actuals and model predictions.

The development of such a neural model is begun by first developing an appropriate layout of neurons, or connections between network nodes. This includes defining the number of layers of neurons, the number of neurons within each layer, and the manner in which they are all linked. The weighted estimating functions between the nodes and the specific training algorithm to be used must also be determined. Once the network has been built, the model must be trained by providing it with a set of historical project data inputs and the corresponding known actual values for
project schedule and/or cost. The model then iterates on its training algorithm, automatically adjusting the parameters of its estimation functions until the model estimate and the actual values are within some pre-specified delta. The specification of a delta value is important. Without it, a model could theoretically become overtrained to the known historical data, adjusting its estimation algorithms until it is very good at predicting results for the training data set, but weakening the applicability of those estimation algorithms to a broader set of more general data.

Wittig [16] has reported accuracies of within 10% for a model of this type when used to estimate software development effort, but caution must be exercised when using these models as they are subject to even greater sensitivities to the training data than are the standard regression techniques used to calibrate more traditional models. They suffer the additional disadvantage of having obscure internals: it is hard to relate the neural net characteristics and parameters to the phenomenology of the projects being estimated. For these reasons, neural net-based software estimation methods are experiencing very slow penetration into project practice.

3.1.6. Dynamics-based methods

Dynamics-based techniques explicitly acknowledge that software project effort or cost factors change over the duration of the system development; that is, they are dynamic rather than static over time. This is a significant departure from the other techniques highlighted in this paper, which tend to rely on static models and predictions based upon snapshots of a development situation at a particular moment in time. However, factors like deadlines, staffing levels, design requirements, training needs, budget, etc. all fluctuate over the course of development and cause corresponding fluctuations in the productivity of project personnel. This in turn has consequences for the likelihood of a project coming in on schedule and within budget—usually negative. The most prominent dynamic techniques are based upon the system dynamics approach to modeling originated by Jay Forrester nearly 40 years ago [17].

System dynamics is a continuous simulation modeling methodology whereby model results and behavior are displayed as graphs of information that change over time. Models are represented as networks modified with positive and negative feedback loops. Elements within the models are expressed as dynamically changing levels or accumulations (the nodes), rates or flows between the levels (the lines connecting the nodes), and information relative to the system that changes over time and dynamically affects the flow rates between the levels (the feedback loops).

Within the last 10 years this technique has been applied successfully in the context of software engineering estimation models. Abdel-Hamid has built models that will accurately predict project cost, staffing needs and changing schedules over time, as long as the initial proper values of project development are available to the estimator [18–21].

More recently, Madachy used system dynamics to model an inspection-based software lifecycle process [22]. He was able to show that performing software inspections during development slightly increases development effort, but decreases later effort and schedule during testing and integration. The overall savings in project effort resulting from that trade-off is a function of development phase error injection rates, the level of effort required to fix errors found during testing, and the efficiency of the inspection process. For most situations, there is an appreciable net payoff (typically 10% savings in effort and schedule), but some situations encounter net losses (e.g. inspecting already defect-free artifacts).

3.2. Software decision support

This section discusses software decision support challenges in terms of the categorization of decision issues into strategic and tactical decision issues identified in Section 2.3. It then identifies a number of emerging opportunity areas in which economic analysis techniques appear attractive for addressing the challenges, such as discounted cash flow, utility theory, multi-objective decision analysis real options theory.

3.2.1. Software decision support challenges

3.2.1.1. Challenges: strategic issues

A major economic analysis challenge at the strategic or enterprise level involves relating software decisions to the enterprise’s bottom line. This can sometimes be straightforward: decisions to automate inventory control can be clearly related to fewer lost sales and reduced inventory and associated carrying costs. However, other aspects are more complex: determining the value of better inventory data to strategic decision-making, or the value of higher quality or speed in developing the software.

Another strategic analysis challenge area involves enterprise architectures and product lines. Conventional wisdom holds that standardizing on interfaces is better than standardizing on software products because of the ability to switch to new best-of-breed products, but there is a shortage of economic tools to validate this for particular situations. Some good frameworks for evaluating product line investments are available (e.g. Ref. [23]), but they fall short of addressing such questions as the optimal breadth of product line to encompass.

Other strategic analysis challenges involve techniques for balancing the risks and rewards of strategic partnerships (one dimension of this is analogous to the “standardizing on interfaces vs. products” question); and for evaluating alternative mixes of strategic improvement investments (process maturity, tools, components, personnel incentives, etc.) [29].

A final strategic analysis challenge involves the use and aggregation of utility functions. For example, an individual manager’s utility function is often highly asymmetric: there
is more negative utility in showing a loss of $1M than there is positive utility in showing a profit of $5M. This tends to stimulate risk-averse behavior and loss of opportunities at higher levels of aggregation, where a couple of $1M losses to generate a few $5M gains are not a problem. Again, some analyses such as this are relatively straightforward, but the general case involving software product and market interdependencies and uncertainties becomes much more complex.

3.2.1.2. Challenges: tactical issues Some tactical issues have analogies to strategic issues, such as the utilities, risks, and rewards of reusing components within a product line. An individual programmer with a 4-week deadline may choose to redevelop a component which he/she knows can be done in 4 weeks, rather than reuse a component which will lead to completion in 2 weeks 80% of the time and in 6 weeks 20% of the time. Again, this simple case is straightforward to address, but more complex cases with component interdependencies and associated uncertainties become considerably more difficult.

Some other tactical economic analysis challenges across a project’s life-cycle involve:

- analyzing what is the best choice of product to develop (or whether to maintain or redevelop a legacy system);
- analyzing the relative value of different product levels of service (performance, reliability, portability, usability) and tradeoffs among them;
- establishing an economic theory of design, including evaluating the benefits of preserving options or late binding;
- analyzing pricing and packaging alternatives: charging by copy, by transaction, by user, etc.:
- determining when to ship a product: balancing the risks of being too buggy or too late;
- determining “how much is enough” prototyping, inspecting, testing, configuration management, COTS evaluation, etc.

3.2.1.3. Challenges: general Both the strategic and tactical areas share further economic analysis challenges. Good data is scarce, particularly with comparable assumptions and definitions across organizations. Stronger techniques for dealing with the various facets of the value of information are needed: timeliness, precision, reliability, accessibility, comprehensibility, decision-relevance, etc. Stronger techniques are needed for analyzing benefits, which lag behind cost analysis techniques. And improved mechanisms for technology transition are needed: many relatively mature techniques are infrequently used, such as present-value and earned-value techniques, and reuse economics models.

3.2.2. Decision support: emerging opportunity areas

The essence of the economic perspective on software engineering is to treat technical decisions as investment decisions and to analyze them as such. For example, deciding which bug to fix given limited time is a choice to commit resources, including time, in the expectation that the decision will create value. Value, in this formulation, is judged in terms of both present and future, certain and uncertain costs and benefits. Value often is measured in monetary terms, but other dimensions can serve as well, such as safety, for example. Similarly, a decision to cancel a project is also an investment decision made in the expectation that it will create value, presumably in the form of reduced future losses.

The axiom that justifies the application of economic concepts to decision making in software engineering is that the final arbiter of success in software engineering, as in all design, is value added as assessed in economic terms, which is to say in terms of what people ultimately value. Economic modeling and analysis in the software domain is not a gratuitous attempt to shoehorn technical problems into an ill-fitting frame, but an attempt to use the language of economics to help to explicate the complex relationships between technical factors and economic costs and benefits, in order to help the decision maker to make those decisions that most effectively contribute to the creation of value. In some cases, imprecise, qualitative analysis is the best that can be done. In other cases, detailed business cases can be constructed. The essence of the approach is in the modeling of the costs and contributions of design decisions.

As we discussed above, the estimation of project costs has been treated extensively using a variety of techniques. In this section, we identify emerging opportunity areas in which economic analysis appears attractive. The modeling approaches that we discuss include decision discounted cash flow (DCF) static analysis, DCF dynamic analysis, decision analysis based on utility theory, multi-objective decision analysis, and real options theory. Our discussion of static and dynamic discounted cash flow and real options follows Teisberg’s excellent account [30].

3.2.2.1. Static discounted cash flow analysis Discounted cash flow analysis is the most widely taught approach to the quantitative evaluation of capital investment opportunities in business contexts [34]. Boehm [6] has discussed the static DCF as a fundamental technique for decision evaluation in software engineering.

Mathematically, the approach involves summing the present values of the estimated monetary costs and benefits (cash flows) that would occur, spread over time (in cash flow streams), if the decision to invest were made. The summation models the idea that the value added as a result of investing is the sum of the benefits minus the sum of the costs. The present value of a cash flow is the amount that an investor would be willing to pay today for the right to a cash flow expected but not certain to be generated by the investment in the future. The present value is less than the nominal amount of the future risky cash flow for two reasons.
First, in a modern economy in which borrowing and lending are easy, making it possible to move cash flows across time, an economically rational investor prefers a given amount of money now rather than later, because it can be used to advantage in the interim. In particular, it could be lent at the “risk-free” rate through an investment in high quality bonds, for a greater future payoff. The present value of a future cash flow is thus discounted in relation to present flows. In particular, it is reduced to the point that if it were presently invested at the risk-free rate it would produce the future payoff precisely. In other words, future cash flows are reduced to reflect the time value of money.

Second, future payoffs from current investments are risky, and any rational investor prefers a sure payoff of a certain amount to an uncertain one. Thus future cash flows are also discounted to compensate the investor for risks incurred. This is done either by discounting each future uncertain cash flow to its certainty equivalent (the amount for which the uncertain payoff could be sold in the market), and then discounting those certainty equivalents to the present time at the risk-free rate; or by using a risk-adjusted discount rate higher than the risk-free rate by an amount that achieves the same effect.

It is now widely understood that static discounted cash flow models are not especially appropriate for modeling the value of complex projects. The basic reason is that it is deficient in its treatment of uncertainty and of the manager’s ability to change course dynamically as new information is obtained. But of course those are precisely the characteristics of most strategic investments, in general, and of investments in many software assets, in particular. Instead, the approach is based on a single projection of cash flow streams. Risk and uncertainty is reduced to the risk premium added to the risk-free discount rate.

The fundamental problem with this modeling approach is that it does not enable modeling of the resolution of uncertainty over time; nor of the manager’s flexibility to make decisions to respond as conditions unfold. It also requires a single subjective estimate of future cash flow streams, rather than representing the range of outcomes that could occur under various future states of the world. By contrast, software engineering is fundamentally an activity of decision making over time usually in the face of significant uncertainties. The mismatch is fairly dramatic.

3.2.2.2. Dynamic discounted cash flow analysis Dynamic discounted cash flow analysis seeks to overcome the limitations of the static DCF modeling approach. It begins with the observation that different cash flow streams within a project can be subject to vastly different risks, and should thus be discounted for risk at varying rates. For example, the costs incurred by an investment might be significantly less certain than they payoffs that are generated [30]. Next, it uses decision trees to model explicitly the different courses that the future could take. A fundamental problem with the approach is that it requires the use of multiple discount rates, because the risks to the cash flows represented in the decision trees are different under different future conditions. For example, the risk to the returns on the investment in the early engineering phase of a project is probably higher than to the returns on investments made in the later production phases—by definition after the key risks have been resolved. Such interdependencies and other dynamic phenomena cause the estimation of multiple discount rates to be fraught with difficulties, but the conceptual framework is highly useful for approximate reasoning.

3.2.2.3. Decision analysis (utility theory) Utility theory measures value not in terms of expected monetary payoffs from a risky investment, which is what the market cares about, but in terms of the expected utility of the risky payoffs to an individual decision-maker. Utility theory yields a subjective evaluation of the worth of a risky asset to a decision maker who has a given tolerance for exposure to risk under different circumstances. A small company whose future depends on the success of a given product might, for example, value its risky payoff less than would be indicated by a discounted cash flow analysis, which, in a sense, reflects the “market’s” posture toward risk. Sections 3.2.2.1 and 3.2.2.2 also discussed how individuals’ asymmetric utility functions might lead to risk-averse behavior which may be suboptimal at higher levels of aggregation. A considerable amount of work has been done lately in developing a theory of engineering design based on utility theory [31].

3.2.2.4. Multi-objective decision analysis The finance perspective on decision making is roughly that the objective is to make money and that essentially all value can be represented in monetary terms, at least in principle. Some people see a problem in this view. They argue that not all values are reducible to money. Consider public safety, for example—an important issue in the software engineering of critical systems. Here value might be measured instead in terms of avoidance of irreversible harm to human lives or the environment.

In some cases, it is possible to reduce non-monetary measures to monetary ones by defining mapping functions. One example would be the mapping of costs in terms of human lives to equivalent economic costs. Such an approach cannot be questioned on ethical grounds; nor are such equivalencies necessarily going to be validated by juries assigning damages. On the other hand, policy-makers often face difficult situations in which such tradeoffs are inevitable. When is the software for a flight avionics computer safe?

If an equivalence is used, it is critical for the analyst to understand that the mapping is often non-linear. For example, at one end of the trade-off curve it might be possible to trade a small amount of safety for significant monetary
savings, while at the other end a small amount of money might purchase a major safety improvement. Thus is does not necessarily makes mathematical sense to employ a simple linear equivalence, such as a life being worth a given fixed number of dollars. The issue of optimizing investments when value is measured in multiple dimensions that are not easily commensurated is the topic of multi-objective decision making.

Multi-dimensional, non-linear trade-offs arise frequently in software engineering. For example, gaining an increment of quality in an ultra-quality systems can be very expensive, e.g. requiring certified conformance to demanding test coverage criteria. The same increment in quality might be purchased through the use of standard software engineering practices at relatively low cost in a low-maturity and low-productivity development organization. Haimes discusses these issues in detail [32].

3.2.2.5. Real options valuation techniques A key problem with both utility theory and discounted cash flow analysis is that they require subjective estimates of the probabilities of uncertain future events. Real options mitigates this problem by appealing to “the market” as a source of more objective estimates. The idea is to treat the managerial flexibility to intervene in a project—often created by wisely structuring the project or product in the first place—as being in the form of options. The key idea is that these options can be valued by applying adapted forms of arbitrage-based options—pricing formulae, such as those originally developed to price financial options, and for which Merton and Schols won a Nobel Prize.

The conditions under which the approach can be applied are simple but fairly demanding. The “markets must be complete,” in the sense that the opportunity that the investor has to invest is equivalent in a certain sense to a portfolio of assets that are already traded in and whose risk has thus been priced by the market. Moreover, the analysis has to have a good estimate of the current value of the asset that would be acquired if the decision to invest were undertaken. In this case, the present value and in particular the variance in future values of the asset in question (the uncertainty) are determined without appeal to subjective estimates. Instead, the judgement of the market is used to judge risk.

When these conditions hold, the problems involved in using dynamic NPV and related approaches can be avoided. Simple parameter values derived from market data suffice. The critical issue here is to be able to distinguish between risks that have been priced (implicitly) by the market, “market risks,” and those on which the market has no opinion, namely “private” or “technical” risks.

Baldwin has appealed to real options concepts to develop a theory of the structure of the computer industry based on a dynamics that is driven by the options value of information hiding modules [33]. To the best of our knowledge, she was the first to connect a traditional concept from the realm of software architecture explicitly to an economic model, one based on real options.

4. Future trends

The rapidly increasing economic significance of information technology is creating comparable increases in demand for information and software economics analysis techniques. The most significant challenge today is to close the gap between software cost analysis techniques and software benefits analysis. Both are needed for effective decision making, but currently software-oriented benefits analysis techniques lag those for software cost analysis.

Software cost analysis techniques require a strong understanding of software phenomenology and a reasonable level of understanding of econometric techniques (e.g. parametric modeling and analysis; probability and statistics). But software benefits analysis also requires a strong understanding of market factors (where to expect what kinds of economies of scale, network externalities, etc.).

In the past, software engineering researchers have been relatively content to “tend their own garden” and concentrate on the software phenomenology they know best. But the pressures for improved cost–benefit and return on investment analyses are causing more software researchers and business-analysis researchers to come together to integrate their knowledge and tools into more effective capabilities, not just for analysis, but also for more effective software management.

As just one example of the potential benefits of such capabilities, consider the prospect of monitoring both the cost and benefit analysis assumptions underlying the business case for a software product as the product is being developed, and flagging events that cause the assumptions to become less valid or invalid. It is not extreme to suggest that such an early-warning system could cause non-viable software projects to be detected and cut off in half the time it currently takes. Applying this 50% saving to the Standish Group’s estimate of $81 billion lost in cancelled software projects per year [24] creates a $40 billion annual savings, a worthy target for improved software economics techniques.

References