Software Design Decisions as Real Options

KEVIN SULLIVAN
University of Virginia
sullivan@cs.virginia.edu
http://www.cs.virginia.edu/∼sullivan
Tel. (804) 982-2206
FAX: (804) 982-2214

PRASAD CHALASANI
Los Alamos National Laboratory
and Carnegie Mellon University
chal@cs.cmu.edu
http://www.c3.lanl.gov/∼chal

SOMESH JHA
Carnegie Mellon University
sjha@cs.cmu.edu
http://www.cs.cmu.edu/∼sjha

June 3, 1997; Revised June 24, 1997

University of Virginia Department of Computer Science Technical Report 97-14
Submitted to IEEE Transactions on Software Engineering

Abstract

Despite their status as foundational concepts in software engineering, many software design decision-making principles and heuristics, such as information hiding and the delaying of design decisions, are still idiosyncratic, ad hoc, poorly integrated and not clearly based on any sound theory. In this paper, we develop an economics-based approach to providing a firmer foundation for software design decision-making heuristics. We start with the premise is that many software design decisions are essentially about when if ever to make irreversible but delayable investments of valuable resources in software assets of uncertain value. This formulation reveals an analogy between software design decisions and *real options*, which are capital investment analogs of financial call options, for which there is a well-developed theory and body of knowledge. In particular, the theory of real options captures precisely the idea that there can be significant value in having flexibility to wait for better information before committing valuable resources to develop or obtain assets. The options-theoretic nature of many software design decisions allows us to bring option theory to bear on an analysis and refinement of critical, widely employed software design decision-making heuristics.

1 Introduction

In design situations of significant complexity, the utility of precise numerical and analytical approaches can break down, leaving engineers largely dependent on heuristic guidelines [29]. Heuristics are essential intellectual tools of the system architect. One problem with heuristics, however, is that it's hard to evaluate their validity owing to the informality with which they are stated. Heuristics such as "Simplify, simplify, simplify," might require little validation. In other cases, however, validity is not so clear; and in such cases, we should seek an understanding of the rational basis for our belief in the validity of our heuristic guidelines.

Should a software designer "Always write a specification?" "Always use information hiding?" "Delay design decisions until they are forced, so as to have the best possible information at the time they are made?" In general, we need to distinguish good heuristics from inadequate ones, and to understand the ranges within which heuristics are valid.

Because software design problems are complex, software engineers tend to depend heavily on a variety of critical and widely accepted design heuristics. Key concepts include information hiding [26] ("hide design decisions that are likely to change"); program families [25] ("delay making design decisions that distinguish sub-families"); spiral software development processes [4] ("attack the greatest risks first"); delaying decisions [16, 36] ("each design decision locks in upstream decisions and constrains downstream decisions"); prototyping [4] ("spend a little early to determine the best course of action"); iterative enhancement [1] ("develop a flexible, operational system early then add capabilities incrementally"); and reuse [19] ("spend extra to design assets so that you can amortize their costs over multiple uses").

Unfortunately, many such heuristics at the heart of software design doctrine and practice are idiosyncratic, poorly integrated, not clearly based on or justified by any sound theory, and, we suspect, suboptimal in many situations. Concepts such as information hiding, while of the utmost importance, are presented in hard-to understand terms that obscure the rational justifications for such concepts.

This state of affairs has many negative consequences. It makes it unnecessarily hard for software designers to reason effectively about design; for teachers to teach the decision-making criteria of the field as a coherent, well founded body of knowledge; for students to truly grasp these criteria; and for managers, who think in terms of wealth maximization over time, to communicate effectively with engineers, who reason in terms of information hiding, delaying of design decisions, secrets of modules, and the like.

Worse yet, the lack of a rational basis for heuristics denies us a satisfying intellectual grasp of the deep connections among critical software design concepts. Does a common structure underlie the ideas of program families, information hiding, delaying of design decisions and architecture, for example? Not having a good answer to such a question ultimately leads to unnecessary problems in cost, quality, timeliness and pain involved in engineering software.

Is software design an inherently opaque subject that demands uniquely strange, difficult and vague concepts? We suspect that the answer is no, and that the opacity of current software design doctrines is largely attributable to shortcomings in the theoretical foundations of our decision-making heuristics—insofar as there is any real theory at all.

One approach to providing a sound basis for decision-making is to appeal to economics, as advocated notably by Boehm [3]. Such an approach recognizes that software design decisions concern the expenditure of valuable resources (time, money, memory, etc.) in the face of uncertainty over future payoffs. The economics approach seeks to make decisions amenable to analysis using techniques from the microeconomics area of *decision-making under uncertainty*.

It is clear that an economics approach to software design decisions is plausible. Design is well understood to be an anticipatory activity that involves projections of an uncertain future and consideration of resource expenditures at every step. For example, the general area of software architecture focuses intensely on the question of the ways in which a system should be designed so as to accommodate "likely change" at an acceptable cost [30].

Consider in particular the information hiding criterion for designing the modular structure of software. The idea is, again, to invest in modules that hide aspects of a system that are viewed as likely to change independently [26, 28]. The costs of a modularization decision includes the cost of designing, implementing, validating, verifying and documenting an interface; the lost opportunity to use the secret of a module directly; and the downstream cost of program restructuring if the future turns out to be other than as anticipated. Benefits include reduced future evolution costs; lower project costs owing to proper abstraction and decomposition; the flexibility to use a module in multiple systems; and the flexibility to produce related systems by varying design decisions independently. Modularization decisions can thus be viewed as decisions to invest resources in the face of uncertainty over what changes are likely, and thus over the future profitability or payoffs of the investment decisions.

In this paper we present the theory of real options as a tool for analyzing and evaluating such critical software design decision-making heuristics as information hiding and delaying of design decisions. The idea is not that we should calculate options values in analyzing specific design decisions, although that might be useful in some cases. Rather, we use options concepts as *intellectual tools* to help us to understand, evaluate, and improve the heuristics that we employ in making design decisions.

The justification for this approach lies in a two-step analogy between financial options and design decisions. First, we will argue that many software design decisions can be viewed as as decisions about when if ever to make irreversible but delayable capital investments in assets of uncertain value. Second, we appeal to an analogy between capital investment decisions and financial call options—an analogy that has been the focus of important recent research in finance and capital investment [8, 9, 10, 13, 22, 34].

Briefly, a call option confers upon its holder the right but not the obligation to purchase assets (such as stocks) at set prices for certain periods of time. The capital investment manager, and we now claim the software designer, is in an analogous situation of having the prerogative but not the obligation to invest resources in "real" assets, such as power plants or user's manuals.

To the extent that the analogies hold, our approach allows us to draw upon the well developed theory and body of knowledge about options in reasoning about software decision-making heuristics. By formulating software design decision-making in financial terms generally, and in terms of real options in particular, we believe that we make progress toward establishing rational foundations for understanding, improving and perhaps even generating important software design heuristics. We hope that this line of research might ultimately help us rationalize our discourse on software design by showing that important heuristics can be seen as manifestations of a common underlying options-theoretic (or other advanced financial) structure. In this paper we shall begin by presenting the basic concepts in options-theoretic terms.

To the best of our knowledge, we are the first to connect real options to the problem of software design decision-making. One of the authors presented an earlier paper outlining the basic idea at the 1996 Second International Software Architecture Workshop [32]. Our work also appears to be distinguished in two additional ways. First, we emphasize mathematical rigor in developing our arguments. Second, seek not so much practical, finance-based tools for direct application in project management, but rather theoretical explanations and tools for reasoning about the highly abstracted heuristics on which software designers depend so heavily.

The rest of this paper is organized as follows. Section 2 discusses the net present value (NPV) approach and its shortcomings. Section 3 provides an informal introduction to the area of financial call options and how they relate to software design decisions. Section 4 provides the requisite mathematical background for the remainder of the paper. Section 5 formally defines some basic options terminology, and also presents known results on the optimal exercise of call options. Section 6 describes the basic idea of real option theory and how to relate it to software design decisions. Section 7 presents an example showing, in more detail, how the real options approach can be applied in reasoning about a fictional problem of deciding when if ever to restructure a software system to improve its information hiding characteristics. Section 8 illustrates the theory of the preceding section by means of a simple numerical example. Section 9 presents a number of qualitative design heuristics and insights derived from the options view of software design. Section 10 ends with a summary and concluding thoughts.

2 Net Present Value

The traditional economic approach to analyzing software engineering decisions focuses on so-called *net present value* (NPV). The financial aspects of Boehm's seminal article and book on software engineering economics [3], emphasize NPV, for example. Boehm does address uncertainty over the present state of nature using the concept of the expected value of information, but he does not address strategies for responding to uncertainty about the future. The need to reason about and respond to uncertain futures is critical to the software designer, and is the dimension of uncertainty that options theory addresses directly.

More recently Favaro has emphasized the key role that financial analysis can play in software engineering—in the area of software reuse, in particular [11]. Favaro rightfully criticizes analysis techniques that have been used to reason about investments in reuse but that are known to have serious shortcomings in relation to the NPV approach [5]. Favaro then settles on NPV as the most appropriate investment analysis technique.

Under the traditional NPV approach, you analyze an investment decision by first calculating the present value of the income stream that the investment will generate. If future income is uncertain, you compute the expected present value, which weights the possibilities by their estimated likelihoods. Next you calculate the expected present value of the stream of expenditures required to implement the decision. The NPV is the first quantity minus the second: the value of the expected income minus the expected outflow. You then employ the traditional NPV rule taught to every business student: invest if and only if the NPV is positive [5].

A key idea behind the NPV approach is that benefits received in the future should be discounted according to an assumed interest rate that generally depends on both macroeconomic factors and on the riskiness of the proposed project. By discounting, we mean that a dollar that is to be received tomorrow is worth less than a dollar held today. The reason is that, with a non-zero interest rate, you could invest less than a dollar today to get a dollar tomorrow. Tomorrow's dollar is worth today what you would have to invest today to have the dollar tomorrow. A dollar received in two days is worth even less today because, with daily compounding, even less would have to be invested today to yield that dollar in two days.

Unfortunately, the simple NPV rule is often suboptimal, because it is founded on a faulty assumption. It views the investor's (or software engineer's) decision as being now-or-never, in the sense that if the investment is not made now, the opportunity to invest is lost forever. If the only possibilities available to the investor are to invest now or invest never, then she is justified in following the traditional NPV rule. However, many investment decisions, including many of those required to effect software design decisions, can be delayed without forgoing the prerogative to invest later should conditions turn out to be favorable.

A decision not to write a user's manual today, for example, can be reversed tomorrow. In the face of uncertainty over the future value of such a document, one might decide to delay investing valuable resources in it. However, should it become clear that the benefits of having the manual are likely to outweigh the cost to produce it, then one could reverse the decision to delay investing, and invest the resources needed to produce the document.

Indeed, delaying software design decisions has long been recognized as an important software design strategy [16]. The theory of options suggests why this might be so. Unlike a decision to delay investing, a decision to invest is irreversible. Once time, money and materials are invested in such a document, those resources cannot be recovered by reversing the decision to invest. The manual might have some scrap value, but the decision to scrap it isn't a decision to reverse the investment, but rather a subsequent decision to sell off the acquired asset at its market value, whatever that might be. In many software design cases, the scrap value of a software artifact is essentially zero. The non-reversibility of decisions to invest in software assets is quite profound.

This asymmetry in the reversibility of decisions to invest and to delay investing brings into play several strategies beyond traditional NPV that should be considered in attempting to make optimal decisions, and which shed light on the situation of uncertainty about the future in which the software designer often finds herself. Specifically, we see that an investor's or designer's decisions can be made *contingent* on what kind of future actually unfolds. For instance, an investor can wait for a month, and decide not to invest if the next month reveals information indicating that the likely payoff would be negative. Or perhaps after a month the uncertainty over the payoff is less, even if the expected payoff is the same; and that might be enough to change the wary investor's mind.

The theory of options shows that this type of wait-and-see strategy can have a larger expected payoff than investing right away even when discounted to the present time. Even when the NPV rule indicates that an investment has a positive expected value, it can be suboptimal to invest right away. The basic problem is that the NPV approach doesn't properly account for the combination of the factors of uncertainty, irreversibility and one's ability to postpone decision-making.

Of course, if the resources invested in an asset can be recovered in full if conditions turn sour, then a decision to invest is reversible, and in this case it would suffice to view investing as a now-or-never decision: Invest if there's a reasonable chance of a good pay off, and pull your money out later if things don't work out. However, as we noted, expenditures made in capital investments, and in implementing software design decisions, usually cannot be recovered. Investments in software design decisions are thus generally *irreversible* but often delayable.

Similarly, if the future is certain, then NPV is an adequate decision rule. If it is certain that an investment will produce a profit, then there is no sense in not investing. Conversely if an investment is a sure loser, no rational decision maker would invest. The problem is that when the future is uncertain, and when in particular there is a chance that an investment will lose (even if the expected return is positive), then it can pay to wait for better information before you invest.

Of course a decision to delay investing can itself have costs. If a user's manual would provide benefits today—e.g., for test case design—then to delay investing is to forgo those benefits. Delay can have an opportunity cost, too. The optimal decision strategy balances the value of waiting for more information against the cost of not having assets now. Clearly, naive software design heuristics such as “always write a manual” can be woefully suboptimal.

Rather than viewing investments under uncertainty as now-or-never propositions or as being reversible, the approach we emphasize in this paper stresses the view that companies and software designers alike have *opportunities* to invest, and that they must decide how to create and exploit such opportunities optimally. We view optimal design decision-making as akin to optimal timing of decisions to exercise *options*—i.e., to exploit investment opportunities.

The success of the “real options” view of capital investment is based on the important observation that investment opportunities are analogous to financial *call options.* A call option confers upon its holder the right but not the obligation to purchase an asset at a set cost for a period of time. Call options are options to purchase financial assets such as stocks. The valuation and optimal exercise of such financial options has been an active area of research in finance over the past two decades. More recently, researchers have been applying results from the area of options to capital investment problems, and this has lead to the new field of *real options theory.*

The theory of real options is based on a strong analogy between call options and capital investment opportunities. The analogy is that when a company makes an irreversible capital investment, in effect it "exercises" a call option. That is, it exercises its prerogative to invest resources to obtain a real asset. To make the analogy explicit, such investment opportunities are called "real options." A real option can be seen as an opportunity to in a real, rather than a financial, asset, and as the flexibility that a manager has to delay deciding whether or not to invest [31]. Thus a company's capital investment problem can be viewed as one of obtaining and optimally exercising real options.

The central idea in this paper, then, is to view software design decision-making as capital investment decision-making and to better understand design by employing concepts from the theory of real options—a theory that in turn borrows heavily from the theory of financial call options. We present this work as a step toward theoretical foundations for software design heuristics based on new concepts from advanced finance. We do not believe that developing such a foundation will make software design easy. Estimating the relevant parameters of actual design situations will remain hard or impossible in real projects. Even with plausible estimates of future benefits and likelihoods, computational complexity and other barriers might prevent instrumental application of such theories. We are not proposing a silver bullet.

Indeed, we do not even intend primarily to present options theory as an analytical tool—although it might be useful in that mode in some cases—but rather an intellectual tool to help us to think better about complex design situations. We see a number of benefits from this line of research. First, like the Navier-Stokes fluid flow equations, a good theory is intellectually satisfying and useful even if it can't always be applied directly. Second, we suspect that many software designers are operating with design rules of thumb that are demonstrably suboptimal, and that basing heuristics on well developed theories can help designers to tune and better understand their heuristics. Third, giving software engineers knowledge of key concepts in finance in the form of financially-based heuristics could help to bridge a serious communication gap between software engineers and capital investment managers.

In this paper we make and present evidence for three claims. First, many software design decisions amount to decisions about capital investment under uncertainty, irreversibility and delayability. Second, we can understand key software design principles as ad hoc and idiosyncratic rules that implicitly reflect the capital investment character of software design and that in many cases tacitly embody real-options-based strategies. Third, appealing to advanced concepts in finance appears to offer some promise to help us to simplify, unify, rationalize, generate, and improve important software design decision-making heuristics.

3 Informal Overview of the Options Approach

In this section we show informally by means of a simple example the principal ideas behind the options approach to software design decisions. A more complete, formal treatment appears in the remaining sections.

Let us suppose that an engineer is considering remodularizing a large software system to impose a new information hiding interface (c.f., Griswold [15]). Such an interface is intended to hide a design decision, or "secret," that is judged likely to change and that other modules needn't "know" about. The expected benefit of such an information hiding approach is that it keeps the cost of change down by limiting dependences on aspects of a system that are likely to change.

Let us estimate that restructuring will cost 1600 dollars. Hereafter we express all resource figures in dollars. Since this is a costly investment, the engineer must ponder carefully the potential benefits from the new design before going ahead with it. We formulate the problem as having to decide when if ever to perform the restructuring. The decision is amenable to an options analysis because, like many software design decisions, it is characterized by the combination of uncertainty about future outcomes, irreversibility, and delayability.

First, the benefits of restructuring are uncertain, depending on when or whether the anticipated requirements changes actually materialize. Second, a decision to invest would be irreversible: The expense incurred in restructuring the system would be unrecoverable. One could revert to the old system, but the money spent restructuring would be lost. Third, the designer is not forced to make a decision immediately, but has flexibility to postpone deciding, hoping to make a better decision later.

The nature of the uncertainty about the future is critical in this case. If the designer knew for certain exactly how much cost-savings the information hiding interface would yield, then her problem would be simple: Restructure if and only if the future profits discounted to the present time exceeds the cost of 1600. However, the future profit stream from restructuring could depend upon uncertain changes in usage patterns, new hardware, other changes in technology or markets, etc.

Thus the designer might be left with no more than a model of how the future profit stream depends on events that might or might not occur. In this case, at any given time, based on estimates of the lifelihoods of and benefits associated with various outcomes, an *expected value* of the future profit stream discounted to the present time can be computed. For brevity we refer to this expected discounted value as the *expected benefit* of a decision to invest at a given time.

Let us consider a particularly simple model. Suppose that the expected benefit of restructuring immediately is 2200. Furthermore, suppose that the benefit of restructuring one month from now is either 3300 or 1100, but that the outcome depends on how certain events turn out between now and then. After one month, the actual outcome will be known; but suppose that at present we can only estimate that each outcome has a probability of occurring of 0.5. Also, let us assume a discounting factor of 1.1 per month. That is, 1.1 dollars a month from now are worth one dollar today.

A standard software engineering approach to this decision problem would use the traditional NPV rule: If the NPV—the expected benefit at the present time minus the expected cost—is positive then invest, otherwise do not. In our example, the cost to invest is 1600, and the expected benefit at the present time is 2200, so the NPV is $2200 - 1600 = 600$. The conclusion from the NPV rule would thus be to restructure immediately for an expected payoff of 600.

Is that the best policy? We'll see that it isn't. The NPV rule views the investment problem as a now-or-never decision. If there were no possibility of delaying the decision, then the NPV rule would be justified. However, as we said, the engineer has the flexibility to postpone deciding. If she invests today, she runs the risk of losing money if the unfavorable scenario emerges. Instead, she can wait a month and undertake the restructuring only if the situation turns out to be the one in which the benefit is 3300 rather than 1100. Interestingly, the NPV of this strategy *at the present time* is significantly greater than the NPV of investing immediately,[1] at

$$(0.5)(3300/1.1 - 1600/1.1) = 773.$$

This formula represents the idea that the cost to invest in a month is the same in nominal dollars as it is today (1600), but that the value of those dollars is less by a factor reflecting an interest rate of 10% (1.1); and that in a month we stand even odds of being able to pay 1600 for an asset worth 3300 discounted dollars. If the benefit turns out to be 1100 rather than 3300, we wouldn't invest, because to do so would surely lose money. Because a decision to invest today would be irreversible, and because the future value of the investment is uncertain and might be less than the cost, and, finally, because the designer has the flexibility to wait, waiting is a better strategy.

To explain in more depth what options have to do with this problem, we need to introduce some basic options concepts. An American *call option* (on a stock) is a financial contract between the option writer and the option holder that provides the holder the right but not the obligation to acquire a share of stock from the writer at a certain price L, called the *strike price*, by a certain date. If the option holder decides to use the option to buy a share of stock from the writer at some time, she is said to *exercise* the option. By exercising the option at time t, the holder acquires an asset worth S_t, and, to acquire this asset, she pays the exercise cost L. The cost to exercise is L and the expected benefit is S_t. (The price of a stock reflects an expectation about the future value of ownership in the issuing company.) Note that the option holder is not obligated to exercise the option even if $S_t > L$; she may wait until a time of her choosing, even possibly letting the option expire without exercising it.

Clearly, a rational option holder will exercise at a time t only if the market price S_t of the stock exceeds the strike price L of the option. Then the holder can make a profit (or "payoff")

[1] Note that although the term NPV is used to refer to the present value of any strategy, contingent or not, the NPV rule found in corporate finance textbooks considers only the strategy of investing right away. Thus when we refer to the "traditional NPV rule", we mean the rule that only analyzes the NPV of immediate investment.

of $S_t - L$ by exercising the option and immediately selling the acquired stock at the market price. Thus, at any time t, the potential payoff from exercising the option is $\max(S_t - L, 0)$, where the zero represents the decision not to exercise the option. We denote this quantity by $(S_t - L)^+$.

The option holder thus faces the question of when if ever to exercise the option in order to maximize the expected payoff discounted to the present time. This situation that appears to be strongly analogous to the design decision-making problem faced by the software engineer. The cost of 1600 to restructure is analogous to the strike price L of a call option on the new interface. By paying this exercise cost at time t, the engineer will acquire an asset worth S_t, which is the expected present value of the future profit stream (e.g., reduced future costs) from restructuring. If we let subscripts denote time in months, then $S_0 = 2200$, and S_1 is either 3300 or 1100, each with probability $p = 0.5$. Thus, the expected payoff from investing in restructuring at the current time is $S_0 - L = 600$. A month from now, if $S_1 = 3300$, then the payoff is $3300 - 1600 = 1700$, and if $S_1 = 1100$ the engineer would not invest, for a payoff of 0. Thus, the engineer can be viewed as holding a real option: She has the right but no obligation to acquire the expected benefit of S_t by investing L in restructuring at time t. Deciding the best time to invest is analogous to deciding the best time to exercise the real option.

Let us now consider two possible exercise strategies. Since there is no uncertainty after time 1, these are the only two alternative that we need to consider. Strategy 1 is to exercise immediately, i.e., at time 0. The payoff is $(S_0 - L)^+ = 2200 - 1600 = 600$. Strategy 2 is to wait one month, and exercise only if the benefit turns out to be 3300. The expected present value of the payoff from this strategy is, of course,

$$(0.5/1.1)\,[3300 - 1600] = 773.$$

The analogy between options and investment decisions yields new and useful software engineering insights. Because past economics-based approaches to software engineering have focused on traditional NPV analysis, they have tended to ignore the need to respond strategically to uncertainty about the future, and they have ignored in particular the value of being able to wait for better information. Options analysis makes this value explicit. In particular, one can define the value V_t of an option at any time t rigorously as the expected present value of future payoffs under the optimal exercise strategy. In our example, the value V_0 is 773 since this is the expected payoff from the best exercise strategy of the two available.

This value can be thought of as the value of having the flexibility to wait. The ability to change one's mind by reversing the decision not to invest, and hence the value of this flexibility, is lost when the option is exercised. In a sense, the option value V_t represents the *opportunity cost* of exercising at time t. It can be shown rigorously that for an American call option, the optimal exercise rule is to exercise when the payoff, $(S_t - L)^+$, is equal to (or greater than) the value V_t, i.e., when $S_t = L + V_t$. This rule says that it is optimal to invest when the benefit S_t exceeds the direct cost L plus the opportunity cost V_t.

This example illustrates how real option theory can help quantify the value of waiting before embarking on an expensive project, such as a major software restructuring. As we will see, the value V_t can be computed easily under simple models of future uncertainty.

Beyond correcting a shortcoming in the NPV approach, the real options approach has the advantage of capturing essential aspects of software design decision-making in a general sense. We are continually faced with uncertain futures, with opportunities to invest, with the right to delay investing, and with costs of both investing and of not doing so.

Having identified the analogy between software design decisions and irreversible capital investments under uncertainty suggests that we use the well-developed body of knowledge and theory of real options to help illuminate software design decision-making heuristics. Moreover, if the analogy between software design decision making and real options is justified, and if the heuristics work, then the heuristics must at least approximate underlying options-based strategies. Making the underlying structure explicit promises intellectual and perhaps even analytical benefits.

4 Mathematical Background

Before describing options and their connection to investment problems more formally, we need some mathematical vocabulary. In much of the following discussion, certain tedious technical conditions and definitions will be omitted.

For simplicity of exposition, we will model future uncertainty by means of a discrete **event tree** of finite depth N, where N represents the maximum number of future time steps (e.g. months, years, etc) that we wish to model. We will often take N to be so large that we can treat it as essentially infinite. Each node in the tree represents a state of the world. The root node is considered to be at depth 0 and represents the present time, i.e., time 0. A node at depth k represents a state of the world at time k; its children are the possible next states at time $k+1$. A typical path in this tree from the root to a leaf (i.e. from time 0 to time N) is denoted by the letters α or ω. We write $\omega^{(k)}$ to denote the prefix of ω consisting of the states from time 0 to time k; $\omega^{(0)}$ represents the empty path. The collection of tree paths ω is called the **sample space** Ω. For our purposes a **random variable** Z is a mapping (or function) that associates with each $\omega \in \Omega$ a real number $Z(\omega)$. A random **process** is a sequence of random variables such as $\{Z_n\}_{n=0}^{N}$, which we will sometimes denote simply by Z_n.

As an illustrative example, it is useful to have the following simple event tree, called the **binomial tree**, in mind. Imagine we toss a coin N times. Each non-leaf node in this tree has two children. If we imagine the tree branching left to right, each of the 2^N paths represents a particular sequence of coin-toss outcomes. On any path ω, for $k = 1, \ldots, N$, the k'th branch is an up-branch if the k'th coin-toss comes up heads (H), and it is a down-branch if it comes up tails (T). For future reference we define the following random variables on this tree: For $k = 1, 2, \ldots, N$, we define $X_k = 1$, if the kth coin-toss is a H and $X_k = -1$ otherwise. We refer to X_k as the **random walk process**. In fact, X_k can be seen as representing a particle that starts at the origin and performs a random walk on the x axis: at time k it moves 1 unit to the right if $X_k = 1$ and 1 unit to the left otherwise. The *position* of the particle at time k is given by the random variable Y_k defined by $Y_0 = 0$, and

$$Y_k = \sum_{i=1}^{k} X_i.$$

With each branch in an event tree, we associate a probability, so that the sum of the probabilities of the branches emanating from a given node is 1. For instance if we have a fair coin in the coin-toss example above, the probability of each branch is 0.5. For any k, and any path ω, the probability of the k-prefix $\omega^{(k)}$, denoted $\mathbf{P}(\omega^{(k)})$, is computed in the obvious way: Multiply the probabilities of the k branches in $\omega^{(k)}$. We define the probability $\mathbf{P}(\omega)$ of the entire path ω in the same way. The **expectation** of a random variable X, denoted $\mathbf{E}(X)$, is defined as

$$\mathbf{E}(X) = \sum_{\omega \in \Omega} X(\omega)\mathbf{P}(\omega). \tag{1}$$

We will need to use the generalized random variable $\mathcal{F}_k$, defined as follows. For any path $\omega \in \Omega$, $\mathcal{F}_k(\omega) = \omega^{(k)}$. In other words, the function $\mathcal{F}_k$ associates with any path $\omega \in \Omega$ its k-prefix $\omega^{(k)}$. It will be useful to think of $\mathcal{F}_k$ as representing the "information up to time k". Thus $\mathcal{F}_k$ represents a "state of the world" at time k. A random variable X is said to be $\mathcal{F}_k$**-measurable** if for any path $\omega \in \Omega$, $X(\omega)$ only depends on $\omega^{(k)}$. More precisely, X is $\mathcal{F}_k$-measurable if for any pair of paths $\omega, \alpha \in \Omega$, $\omega^{(k)} = \alpha^{(k)}$ implies that $X(\omega) = X(\alpha)$. For instance in the coin-toss example, if the random variable H_k represents the number of heads up to time k, then H_k is $\mathcal{F}_k$-measurable, for $k = 1, 2, \ldots, N$. Similarly, the random variable Y_k (the number of heads minus the number of tails by time k) is $\mathcal{F}_k$-measurable. A random process $\{X_k\}_{k=0}^{N}$ is said to be **adapted** if for $k = 0, 1, 2, \ldots, N$, X_k is $\mathcal{F}_k$-measurable.

The concept of conditional expectation is an important one for this paper. Let us imagine we are in a particular state of the world at time k, represented by the value of the random variable $\mathcal{F}_k$. In other words, we are on some path ω and we know $\mathcal{F}_k(\omega) = \omega^{(k)}$. Now suppose we want to compute the expectation of some random variable X *given* that we are in state $\mathcal{F}_k$. Clearly this expectation will in general be different from $\mathbf{E}X$, and will depend on $\omega^{(k)}$. For example, in our coin toss example, suppose X is the random variable Y_n. If $\omega^{(k)}$ consists only of heads then the expectation of Y_n given $\omega^{(k)}$ will be higher than if $\omega^{(k)}$ contained only tails. We compute the expectation of X given $\omega^{(k)}$, in a manner similar to expression (1): The difference is that we take the weighted sum of $X(\alpha)$ only over paths α such that $\alpha^{(k)} = \omega^{(k)}$, and we only weight each term $X(\alpha)$ with the product of the probabilities of the branches of α that are taken *after* time k. This probability-product is simply $\mathbf{P}(\alpha)/\mathbf{P}(\omega^{(k)})$. Then the **conditional expectation** of X given $\mathcal{F}_k$ is denoted $\mathbf{E}(X|\mathcal{F}_k)$ and is defined as the

($\mathcal{F}_k$-measurable) random variable that maps any path $\omega \in \Omega$ to

$$\frac{\sum_{\substack{\alpha \in \Omega \\ \alpha^{(k)} = \omega^{(k)}}} \mathbf{P}(\alpha) X(\alpha)}{\mathbf{P}(\omega^{(k)})},$$

which is just a form of the familiar Baye's rule. The conditional expectation $\mathbf{E}(X|\mathcal{F}_k)$ can be thought of as the expectation of X given that we have all the information up to time k, or given the state of the world at time k.

For a random variable X, it is customary to write $\{X = x\}$ to denote the set of paths ω such that $X(\omega) = x$. For any set of paths A, I_A is a random variable called the **indicator function** for A, and is defined as

$$I_A(\omega) = \begin{cases} 1 & \text{if } \omega \in A, \\ 0 & \text{otherwise.} \end{cases}$$

A **stopping time** τ is a random variable taking integral values in the range $[0, N]$, such that for each $k = 0, 1, \dots, N$, $I_{\{\tau = k\}}$ is $\mathcal{F}_k$-measurable. In the coin-toss tree, an example of a stopping time is

$$\tau(\omega) = \begin{cases} \min\{k \geq 0 : \omega^{(k)} \text{ contains 3 Heads}\} & \text{if such a } k \text{ exists,} \\ N & \text{otherwise.} \end{cases}$$

This stopping time can be viewed as specifying the following rule: stop when the coin has landed heads 3 times. Note that if we happen to stop at time k on a path ω, i.e., $\tau(\omega) = k$, then for *any* path α with $\alpha^{(k)} = \omega^{(k)}$, we have $\tau(\alpha) = k$. Thus a stopping time is a non-clairvoyant decision rule of when to stop, and in this sense models real-world decisions that must be made in the absence of information about the future. We remark that “stopping” is just a convenient metaphor that could represent any action for which we are studying decision rules.

5 Financial Options

We now describe some basic concepts in option theory. For further details we refer the reader to Hull's introductory text [18]. For a rigorous treatment, consult Merton's seminal work [23].

The simplest kinds of options are call options. An **American call option** on a certain stock is a financial contract with the following features: it gives the holder of the contract the *right* but not the obligation to buy a share of the stock at a fixed price called the **strike** (or **exercise**) price L from the writer (seller) of the contract, on or before a certain **expiration** date of T time units. The holder thus has the “option” of deciding whether or not to exercise the contract, i.e., demand a share of stock from the contract writer at the strike price L. This is why the contract is called an option. When the option is exercised or the option expires, the option ceases to exist. Thus option exercise is irreversible.

The variable underlying the option is the price per share of the stock specified in the contract (hereafter the “stock price”). It is common in finance to model stock prices as continuous-time stochastic processes that follow definite trends that are disturbed by Brownian motion. A Brownian motion process is commonly used in physics to represent the motion of a particle that is subject to a large number of molecular shocks and that might also be subject to forces exerted by a field in which the particle is embedded. The field accounts for the overall trend; the Brownian motion for a superimposed uncertainty.

This continuous-time process can be approximated in a rigorous sense by a discrete-time process called the **binomial model** [7], defined as follows. The stock price process can be visualized in terms of the binomial tree introduced in the previous section. The life T of the option is divided into N time steps of length $\Delta t = T/N$, and time k in this discrete model corresponds to continuous-time $k\Delta t$. The stock price at time k is S_k, so that S_0 is non-random, and S_k for $k > 0$ is random. At each time step, the stock price either moves up by a factor u or down by a factor $1/u$. In terms of coin-tossing, we can associate an up-tick of the stock price with the coin landing H, and a down-tick with the coin landing T. In terms of the random variable Y_k introduced earlier (the number of heads minus the number of tails by time k) , the stock price process S_k is given by

$$S_k = S_0 u^{Y_k}, \quad k = 0, 1, 2, \dots , n.$$

The probability of an up-tick is p; the probability of a down-tick is $q = 1-p$. When parameters p and u are chosen appropriately, it can be shown that the binomial model for the stock price approaches the above continuous-time model as $N \to \infty$ [7].

In order to discount future cash flows to the present time, we will need to assume that money can be borrowed or lent (for example, via a bank or government bond) at a risk-free interest rate of r. Thus a dollar lent or borrowed at discrete time k is worth $R = 1 + r$ dollar at time $k+1$. It is common to refer to R as a **discount factor** since a dollar at time k, discounted to the present time (i.e. time 0) is worth $1/R^k$.

Now let us return to the description of the American call option in terms of the binomial stock-price model. It is clear that the holder should not exercise the option at time $k \leq N$ if $S_k \leq L$. On the other hand, if $S_k > L$, the holder might but is not obligated to exercise the option; and if she does, the option writer is obligated to sell her a share of stock at the strike price L. The holder could then immediately sell the share in the market at S_k, and make a profit of $S_k - L$. Thus the profit that can be realized from an American call option at time k is $\max(S_k - L, 0)$, which we refer to as the **payoff** G_k from the option. Again, for any real number x it is standard notation to write x^+ for $\max\{x, 0\}$, so we can write the payoff as:

$$G_k = (S_k - L)^+. \tag{2}$$

Since the option holder never takes a loss from holding an option, it has a value for the holder if there is any chance that it might ever be exercised for a profit. It is therefore not surprising that the holder must pay a certain price to the seller for owning the option. Options have value, and trillions of dollars worth of this and many other kinds of options are traded daily in world financial exchanges.

What is the best exercise strategy for the holder of an American call option if she is still holding it at time k? As mentioned in the last section, any non-clairvoyant exercise strategy can be described by a stopping time τ. For instance the strategy of exercising immediately (at time k) is defined by the constant stopping time $\tau = k$. As another example, the strategy: "exercise when the stock price exceeds a certain threshold λ", is described by the stopping time

$$\tau = \min(\{N\} \cup \{m \in [k, N] : S_m \geq \lambda\}).$$

If the holder of the option exercises immediately, she will obtain a payoff $(S_k - L)^+$. If she exercises at some future time $m > k$, the payoff would be $(S_m - L)^+$, which is worth $(S_m - L)^+/R^{m-k}$ at time k. Depending on the value of S_m, this could be smaller or bigger than the payoff $(S_k - L)^+$ from immediate exercise. The stock price S_m at the future time m is of course uncertain and cannot be predicted.

For a given exercise strategy (stopping time) $\tau \geq k$, the expected present value $V_k^{(\tau)}$ can be computed as follows. The payoff upon exercise at time τ is $(S_\tau - L)^+$, which is worth $(S_\tau - L)^+ R^{k-\tau}$ at time k. Therefore the expected present value of the payoff from this exercise strategy, given the information up to time k is

$$V_k^{(\tau)} = \mathbf{E}\left((S_\tau - L)^+ R^{k-\tau} \middle| \mathcal{F}_k\right), \quad \tau \geq k.$$

Our option holder would of course want to choose τ so that this expectation is maximized. We denote this maximum by V_k:

$$V_k = \max_{\tau \geq k} V_k^{(\tau)}. \tag{3}$$

Thus the value V_k, which we loosely refer to as the "value of the option at time k", is the best expected present value at time k realizable over all possible exercise strategies. We note in passing that in option pricing theory, the "fair value," or fair trading price, of an option is defined by assuming the absence of arbitrage. That is, the fair value of the option is the value at which it can be traded so that there are no opportunities for unlimited riskless profit.

More specifically, fair value is defined as the value of a dynamically updated portfolio that replicates the random fluctuation of the option payoff G_k. The replicating portfolio consists of the underlying asset, namely the stock, and risk-free bonds. It turns out that the arbitrage-free fair value of an American call option is defined just as V_k above, but under a specific artificial probability measure called the risk-neutral measure that is not necessarily the "actual" probability measure. However in the case of real options, as we will see, the underlying asset, which represents expected future profits, only exists as a result of exercising the option, and is not traded independently. This makes the option-replication approach to valuation less compelling for real options. Also, since real options themselves are not traded either, we are not so much concerned with their fair value, as with how to exercise them optimally—i.e., with maximizing the expected discounted payoff under the actual probability measure. For the purpose of determining the optimal exercise policy, therefore, we can take the option value V_k to be defined as above.

Since immediate exercise is a valid strategy at any time, V_k must be at least as large as $(S_k - L)^+$. In fact, if $(S_k - L)^+ < V_k$, this means that the immediate exercise strategy is *not* optimal, and that some other strategy will yield a strictly greater expected present value of payoff under our assumed stock price model. Thus, in this situation it is beneficial not to exercise but to wait. On the other hand, if $(S_k - L)^+ = V_k$, then there is nothing to be gained by waiting, at least under our assumed stock price model. In this case it is optimal to exercise immediately. Indeed it can be shown rigorously that the stopping time τ that achieves the maximum in (3) above is given by

$$\tau = \min(\{N\} \cup \{m \in [k, N] : V_m = (S_m - L)^+\}). \tag{4}$$

Let's look at the optimal exercise rule from a cost-benefit viewpoint. We can think of the strike price L as the "cost" of exercising the option, since this is the price one must pay to obtain a share of stock. Similarly, S_k is the benefit from exercising at time k, since this is the price one would obtain by selling the stock in the market. We just remarked above that it may not be optimal to exercise as soon as the benefit S_k exceeds the cost L. To see this, it is useful to view the option value V_k as representing the value of the choice to exercise. When the option is exercised, the option (and the choice) is killed and this value is lost, so that V_k represents the opportunity cost of exercising the option. Thus exercising the option incurs two costs: the *direct* cost L, and the opportunity cost V_k. From the discussion above, the optimal exercise strategy is to exercise when $(S_k - L)^+ = V_k$, which in cost-benefit terms can be stated as follows:

> *Exercise only when the benefit S_k equals or exceeds the direct cost L plus the opportunity cost V_k.*

This is the viewpoint that we will find most useful in this paper.

The value V_k can be computed for all k by a simple *dynamic programming* procedure (see [6]) as follows. First observe that $V_N = (S_N - L)^+$. This is clear both from formula (3) and from observing that the since option expires at time N, there is no advantage to waiting. Now stepping backward in time on the binomial tree, we compute V_k in any state of the world (given by $\mathcal{F}_k$) using the formula

$$V_k = \max\{(S_k - L)^+, \mathbf{E}(V_{k+1}|\mathcal{F}_k)/R\}. \tag{5}$$

In other words, the option value V_k on a path ω is the maximum of the immediate payoff $(S_k(\omega) - L)^+$ and the expected present value of the option value one time step ahead, given that we have seen the prefix $\mathcal{F}_k(\omega) = \omega^{(k)}$ so far. It can be shown that this backward-recursive formula for V_k and formula (3) are equivalent. And this is true regardless of the specific process that the stock price S_k follows—that is, even if S_k does not follow the binomial model assumed here. This fact will be useful in our application to software engineering decisions, since the analog of S_k in those applications does not necessarily follow the binomial model.

6 Real options

Now let us consider a problem of making an irreversible capital investment in the face of uncertainty. Once again we will assume we are working in a discrete event tree of depth N. Suppose a firm is faced with the decision of whether to invest in a factory for making a new type of disk drive. The investment is irreversible, since the factory can only be used to make these disk drives, and if market conditions turn out to be unfavorable, the firm cannot regain its lost investment. For simplicity let us say that the factory can be built instantaneously, at a cost L, and that it can produce disk drives forever at zero cost. Once the factory is built, say at discrete time k, the disk drives can be sold at the prevailing market price. The future profits from disk-drive sales depend on how the market price evolves, which is uncertain.

Let S_k be the expected present (i.e. time-k) value of these future profits, under a suitable market price model, probability measure, and discounting factor. Thus S_k represents the value of the asset that the firm can acquire by exercising its option to invest L at time k. Alternatively, one can think of S_k as the benefit from making the investment at time k, and L as the cost of the investment. Clearly the firm will not invest in the factory if $S_k < L$.

On the other hand, should the firm invest simply because S_k exceeds L? The traditional NPV rule says "yes." However, as argued by Dixit and Pindyck [9] and others, this rule is flawed since it treats the decision problem as a now-or-never proposition. That is, if there is no possibility of delaying the decision, then the rule is indeed reasonable. However, if the decision to invest can be postponed, then the NPV rule ignores the value of waiting for better information before making the investment.

The option viewpoint is the natural framework in which to quantify the worth of the flexibility of being able to choose between investing now and at a future time. If we interpret the benefit S_k as the stock price, and the direct investment cost L as the strike price, then investing in the factory is analogous to exercising the American call option. Thus the value V_k represents the value of the option to invest, or the opportunity cost of investing, at time k. The reason we think of V_k as an opportunity cost is that when we exercise the option, we lose the right to choose when to invest. In analogy with the above rule for an American call option, the optimal rule for the firm given suitable definitions of S_k and V_k is to invest if the asset value (or expected benefit) S_k exceeds the direct cost of the asset L plus the opportunity cost V_k. This idea is at the heart of the theory of real options.

There is nothing in the above discussion that is specific to capital investments made by a company. The approach applies equally well to any investment situation where (a) there is an expenditure of limited resources, (b) there is uncertainty over the future profitability of the investment, (c) the decision to invest is irreversible, and (d) the decision can be delayed.

As we argued before, many software design decisions satisfy these criteria. In general suppose a software engineer is contemplating whether to commit resources to effect a certain design decision. In terms of the variables introduced above, the direct cost L is the cost to effect the decision. The future profit stream is uncertain, depending on factors such as changes in requirements, hardware, usage-patterns, etc. Once a decision to invest is effected, it cannot be reversed. And investing can often be delayed.

The uncertainty over future benefits can be modeled as before in terms of an event tree. At a discrete time k, the asset value or expected benefit S_k is the expected value, discounted to time k, of the future profit stream that would result if the design were already in place by time k . The option value V_k is the opportunity cost of implementing the design—the value of the lost flexibility of being able to decide when if ever to implement the design.

Therefore, as in the capital investment scenario, the optimal decision strategy for the software designer is to invest in the design when the expected benefit S_k equals or exceeds the direct cost L plus the opportunity cost V_k. We thus have a rigorous way to quantify when it is beneficial to delay a decision to invest in a software artifact, design, design change, etc. In the next section, we illustrate the approach in more detail with an example concerning a decision about whether to restructure a software system to impose a new information hiding interface.

7 Applying Options Thinking to a Restructuring Decision

To illustrate our arguments, we will analyze a simple example of a software design decision: namely, when if ever to invest the resources needed to restructure a software system to improve its information hiding properties. We consider a problem in the domain of software agents on the internet [20, 24, 33].

Agents are autonomous software entities that specialize in certain tasks. Several agents are already in place on the World Wide Web. For instance, information agents can handle queries such as, “What is the lowest cost airplane ticket from Charlottesville to Pittsburgh today?” There are news reading and filtering agents. There are also shopper agents that can search for a good bargain on a compact disc. Given the vast size of the internet, it makes economic sense for agents to use other agents to accomplish their goals. For instance, a financial portfolio management agent could use another agent to obtain company reports.

Thus, each agent has a certain *capability*, and one agent might need to find other agents with needed capabilities. We imagine that the capabilities of available agents are stored in a *capability directory*. In the vocabulary of information hiding, the contents of the capability directory are likely to change and should thus be made the “secret” of a module.

Suppose that a software engineer is deciding whether or not to make the directory contents the secret of a module. Specifically, he is deciding between the following two ways to design this directory:

D: Distributed directory. A copy of the capability directory is hard-wired in the code of each agent. An agent that wants to find another agent with a given capability can consult its local directory at essentially 0 cost. This approach exposes the directory, which is likely to change, to all the agents, and thus does not follow the information hiding design criterion very well. Consequently, whenever an agent is added to the system, the directory in each agent must be changed. The absence of an information hiding module magnifies the cost of changes owing to the distribution throughout the system of dependences on volatile information. We denote the total cost of the code changes required when an agent is added at time n by the random variable D_n.

C: Centralized Directory. There is one designated agent called the *yellow pages* agent that implements the capability directory. All other agents access directory information through an interface of this agent. This approach employs information hiding since the new agent hides the aspect of the system judged likely to change. We let C denote the total cost of initially hard-coding the centralized directory and its associated interfaces. When a new agent is created, the yellow-page agent needs to be changed. Also, an agent requiring a certain capability needs to query the yellow-page agent. We let the random variable C_n denote the total query and update costs at time n.

Which approach should the software engineer choose? Suppose, first, that there is no design already in place and no agents to start with. In this circumstance, when the system has to be built, the engineer does not have the flexibility to delay deciding. The choice among alternatives has to be made for the system implementation to progress. Here, the NPV approach is justified.

In order to compare choices C and D, we can assume that the costs that are common to both approaches are 0. The only relevant costs are C and C_n for choice C, and D_n for choice D. In fact we can view the problem as one of deciding whether or not to use choice C, and express the costs and benefits relative to choice D. There are two quantities of interest when choice C is compared with choice D:

- The direct cost of choice C, i.e., the immediate cost of implementing it, which is $L = C$.
- The monthly profit of choice C relative to D in month n for $n \geq 0$, which is $B_n = D_n - C_n$.

We view the software design problem as an investment decision problem: Should L dollars be invested in choice C? Let us assume a discount factor R. Consider the traditional NPV approach to this problem. To apply it, we first compute the expected present value S_0 (discounted to time 0) of the stream of profits $B_n, n \geq 0$ from the investment:

$$S_0 = \sum_{n=0}^{\infty} \mathbf{E} B_n / R^n, \tag{6}$$

which we refer to as the expected benefit of choice C relative to D at time 0. The NPV of the investment at time 0 is

$$NPV = S_0 - L = S_0 - C.$$

The traditional NPV rule states that if the NPV is positive, then the investment should be made, otherwise not. Thus a reasonable decision rule, when there is no design to start with, is the following:

> *If the expected present value of the future profits S_0 that would flow from choice C exceeds the direct cost C of implementing it, then go ahead and implement choice C, otherwise implement choice D.*

As noted by Boehm [3], this kind of rule is often used by software engineers, implicitly or explicitly.

But now we ask a different question, and one likely to arise given that so much software engineering activity is in maintaining existing systems rather than designing new ones. Suppose there is already a system with several agents in place, that the distributed directory was implemented (choice D), and the software designer is contemplating whether or not to invest in restructuring the system so as to switch to choice C. In addition to the cost C of creating the yellow pages agent, there is a cost C^s of scrapping the distributed approach. Each agent must be changed so that it queries the yellow pages instead of its own local directory. Thus the total direct cost of choice C is $L = C + C^s$. Given these costs, how should the engineer decide?

It is tempting to propose the following rule (compare it with the previous rule):

> *If the expected present value of future profits S_0 that would flow from restructuring exceeds the direct cost of restructuring, L, then go ahead and restructure, otherwise do not.*

However as we noted before, there is a serious flaw in this analysis. It compares only two choices: switching to choice C now or never. In the case where there was no design to start with and one was required, the engineer was forced to decide between choices C and D. There was no flexibility to delay making a decision. The decision was now-or-never, and the NPV rule was appropriate. However, in the restructuring situation the designer has the option to wait in hopes of being able to make a better decision in the future.

The flexibility to wait has value that is lost once the designer exercises the option to restructure. The designer cannot reclaim the resources invested in restructuring simply by changing her mind. Thus, in addition to the direct cost L, there is an additional opportunity cost to investing that represents this loss in flexibility. Therefore, at any time k, the value of the expected profits discounted to time k (i.e., S_k) must be sufficiently higher than the direct cost L to justify switching. In short, the designer should compare the value of investing now (at time 0) versus investing at *all* possible future times. She should really be asking *when* if ever to make the investment in restructuring.

We emphasize again that this situation is similar to that of a manager facing a decision to invest capital in new plant and equipment, which, in turn, is analogous to the decision facing the holder of an American call option on a stock. At any time k, the designer/manager/holder has the right but not the obligation to invest L (the exercise price of the option) to receive a stream of profits with an expected present value S_k (the stock price). Exercising the option kills the investment opportunity, just as in the case of an American option.

Let us define the random variable S_k to be the expected benefit of restructuring at time k (the asset value at time k), i.e., the expected value of the future profit stream resulting from investing at time $k,$ discounted to time k. We computed S_0 above, and the payoff from exercising at time 0 is $G_0 = (S_0 - L)^+$, since one would not invest if $S_0 < L$. Notice that this is the same as expression (2) for the payoff from an American call option on a stock at time 0. S_0 is analogous to the stock price at time 0—thus our choice of notation.

How do we generalize the expression (6) for S_0 to time k? To compute the benefit S_k at time k, we proceed as in expression (6), except that we discount the profits to time k rather than time 0. We also replace the expectation by the corresponding conditional expectation conditioned on $\mathcal{F}_k$. Finally, we only perform the summation from times k to ∞. Thus S_k is given by the following expression:

$$S_k = \sum_{n=k}^{\infty} \mathbf{E}\left[B_n R^{k-n} \middle| \mathcal{F}_k \right]. \tag{7}$$

Note that, unlike S_0, S_k is a ($\mathcal{F}_k$-measurable) random variable. The expected benefit of restructuring to institute an information hiding yellow-pages agent at time k is then

$$G_k = (S_k - L)^+, \tag{8}$$

which is the same as expression (2) for the payoff from a call option. The value V_k of this option represents the value of the investment opportunity, which would be lost if we were to exercise at time k. As described in Section 5, V_k can be computed for any k using dynamic programming. Also, we mentioned that it is optimal to exercise the option when the value V_k equals or exceeds the payoff G_k. Thus it is optimal to switch to choice C when $S_k - L \geq V_k$, or

$$S_k \geq L + V_k.$$

Informally, we should exercise our option to switch to choice C when the benefit S_k is at least as much as the sum of the direct cost L and the opportunity cost V_k. Thus our new design decision rule for the software engineer is the following:

> *If at any time k, S_k, the expected value discounted to time k of future profits that would flow from restructuring, is at least V_k more than the direct costs L of restructuring, then go ahead and restructure, otherwise do not.*

8 A Numerical Example with One-Period Uncertainty

We make our analysis of the restructuring decision concrete by means of a numerical example. Suppose the cost C to restructure is 9000, and the cost C^s of scrapping the distributed directory structure is 1000. Thus the total direct cost L of restructuring is $C + C^s = 10000$.

To keep complications to a minimum, we assume that building the yellow-page agent and scrapping the distributed directories takes 0 time. Each discrete time step in our model represents 1 month. Time n represents the beginning of the n'th month, for $n = 0, 1, 2, \ldots$. Let us imagine that during the current month, or month 0, several new agents will be created, and that the associated updating cost under the distributed approach is $D_0 = 2000$. We assume that the total query/update cost under the centralized approach is $C_n = 500$ at all times n. Thus if we move to a centralized directory at the beginning of month 0, the monthly profit for month 0 would be

$$B_0 = D_0 - C_0 = 2000 - 500 = 1500.$$

Suppose an agent development technology is being deployed this month, and there is a probability $p = 0.5$ that it will succeed and be widely accepted. If it succeeds, several new agents will be created each month, starting with month 1. This outcome is favorable for approach C and we will therefore superscript variables under this scenario with the letter f.

In particular, we suppose that in this situation the total update cost associated with agent creations under approach D, is $D_n^f = 3000$ for all $n \geq 1$. (It becomes expensive to accommodate the changes owing to the inadequate hiding of the aspect of the system that turned out to be volatile.) On the other hand, if the technology fails, very few new agents will be created, a situation which is unfavorable for approach C, since the expense of switching to a centralized directory will not be compensated by the cost-savings of the information hiding restructuring. Thus, we superscript variables in this scenario by the letter u. We suppose that the corresponding update cost under choice D in this case is much lower, at $D_n^u = 400$ for all $n \geq 1$.

Thus from month 1 onward, the monthly profit from restructuring for information hiding would be

$$B_n^f = D_n^f - C_n = 3000 - 500 = 2500, \quad n \geq 1$$

in the favorable scenario, and

$$B_n^u = D_n^u - C_n = 400 - 500 = -100, \quad n \geq 1$$

in the unfavorable scenario, each case occurring with probability 0.5. Therefore, for $n \geq 1$,

$$\mathbf{E}B_n = \mathbf{E}B_1 = pB_1^f + (1-p)B_1^u = 0.5(2500 - 100) = 1200.$$

Our model is represented by the event tree in Figure 1. There are just two possible paths in this event tree, which we denote by ω_f (favorable) and ω_u (unfavorable). Note that for $k \geq 1$ and $n \geq k$, on the favorable path,

$$\mathbf{E}(B_n | \mathcal{F}_k)(\omega_f) = B_n^f = B_1^f = 2500,$$

$$\begin{tikzcd}
 & \begin{array}{c} B_1^f = 2500 \\ S_1^f = 27500 \end{array} \arrow[r] & \begin{array}{c} B_2^f = 2500 \\ S_2^f = 27500 \\ \cdots \end{array} \\
\begin{array}{c} B_0 = 1500 \\ S_0 = 13500 \end{array} \arrow[ru] \arrow[rd] & & \\
 & \begin{array}{c} B_1^u = -100 \\ S_1^u = -1100 \end{array} \arrow[r] & \begin{array}{c} \cdots \\ B_2^u = -100 \\ S_2^u = -1100 \end{array}
\end{tikzcd}$$

Figure 1: *Event tree for the numerical example*

and on the unfavorable path,

$$\mathbf{E}(B_n|\mathcal{F}_k)(\omega_u) = B_n^u = B_1^u = -100.$$

Assuming a monthly discount factor of $R = 1.1$, let us calculate the benefit of switching at time k. For $k = 0$, we use expression (6) to compute

$$\begin{aligned}
S_0 &= \sum_{n=0}^{\infty} \mathbf{E}(B_n)/R^n \\
&= B_0 + \sum_{n=1}^{\infty} \mathbf{E}(B_1)/R^n \\
&= B_0 + \mathbf{E}(B_1)/(R-1) \qquad (9) \\
&= 1500 + 1200/0.1 \\
&= 1500 + 12000 = 13500.
\end{aligned}$$

For $k \geq 1$, from expression (7), the benefit from switching at time k in the favorable scenario (or on the favorable path) is

$$
\begin{aligned}
S_k^f &= \sum_{n=k}^{\infty} \mathbf{E}[B_n|\mathcal{F}_k](\omega_f)R^{k-n} \\
&= \sum_{n=k}^{\infty} B_1^f R^{k-n} && (10) \\
&= \sum_{n=0}^{\infty} B_1^f / R^n && (11) \\
&= B_1^f \frac{R}{R-1} && (12) \\
&= 2500(1.1)/0.1 = 27500,
\end{aligned}
$$

which is bigger than the direct cost $L = 10000$. Thus in the favorable scenario the expected benefit of switching is always greater than the cost. Similarly,

$$S_k^u = B_1^u \frac{R}{R-1} = -100(1.1)/0.1 = -1100, \quad k \geq 1, \tag{13}$$

which is smaller than the direct cost $L = 10000$, so in the unfavorable scenario the expected benefit of switching is smaller than the cost. Note that from the definitions of S_0, S_1^f and S_1^u it follows that

$$S_0 = B_0 + (p/R)(S_1^f + S_1^u). \tag{14}$$

Given this model, should the software designer invest $L = 10000$ and restructure in order to switch to design C now, or would it be better to wait for a month and invest only if the situation favors a switch, i.e. only if the new technology succeeds, making it certain that many changes will be needed? Since there is no uncertainty beyond the first month, these are the only two strategies worth considering.

We first approach this question by computing the net present value of these two strategies. The NPV of Strategy 1 is

$$NPV(1) = S_0 - L = B_0 + \mathbf{E}(B_1)/(R-1) - L = 13500 - 10000 = 3500. \tag{15}$$

Since the NPV is positive, the NPV rules indicates that the designer should go ahead with the investment. This reasoning is flawed, as we pointed out before. It ignores an opportunity cost: that of waiting for information and keeping open the possibility of not investing if the technology fails. This opportunity is lost by investing now.

Let us calculate the net present value of investing under Strategy 2: Wait one month, and invest in switching to the yellow-page agent only if the technology succeeds. Since the technology succeeds only with probability $p = 0.5$, the net present value of Strategy 2 is

$$\begin{aligned} NPV(2) &= (p/R)(S_1^f - L) \\ &= (p/R)\left(B_1^f \frac{R}{R-1} - L\right) = (0.5/1.1)(27500 - 10000) = 7954, \quad (16) \end{aligned}$$

which is significantly greater than the NPV of investing immediately. This clearly shows that it is better to wait.

We now approach the question by computing the value V_k of the investment opportunity at times $k = 0$ and $k = 1$. The payoff if we exercise our option to invest at time k is given by $G_k = (S_k - L)^+$ which is identical to the expression for the payoff from an American call option. Since there is no uncertainty after time 1, it is easy to see that $V_k = G_k = (S_k - L)^+$ for all $k \geq 1$. In particular, if the technology succeeds, the option value at time 1 is

$$V_1^f = (S_1^f - L)^+ = (27500 - 10000)^+ = 17500, \quad (17)$$

and if the technology fails,

$$V_1^u = (S_1^u - L)^+ = (-1100 - 10000)^+ = 0. \quad (18)$$

From the backward recursion (5) we conclude that

$$\begin{aligned} V_0 &= \max\{G_0,\ (1/R)\mathbf{E}(V_1)\} \\ &= \max\{G_0, (1/R)(pV_1^f + (1-p)V_1^u)\} \\ &= \max\left\{(S_0 - L), (p/R)(S_1^f - L)\right\} \quad (19) \\ &= \max\left\{(B_0 + \mathbf{E}(B_1)/(R-1) - L)^+,\ (p/R)\left(B_1^f R/(R-1) - L\right)^+\right\} \quad (20) \\ &= \max\{3500, (1/1.1) \times 0.5 \times (17500 + 0)\} \\ &= \max\{3500, 7954\} \\ &= 7954. \quad (21) \end{aligned}$$

Notice that the values 3500 and 7954 in the $\max$ above are exactly the NPVs of strategy 1 and strategy 2, respectively. Also, $V_0 > G_0 = 3500$, so it is not optimal to invest right away. However, after 1 month, if the technology succeeds, $V_1 = G_1 = 17500$, so it is then optimal to invest at that time. Thus we have shown in two different ways that strategy 2 is optimal. In general when the uncertainty lasts for several periods, the approach of computing NPVs for the exponentially many possible strategies is impractical. The second dynamic programming approach from option pricing theory would be the method of choice.

9 Qualitative Design Principles

In previous sections we showed how to view a software design decision as a decision about when if ever to make an irreversible capital investment in the face of uncertainty. We presented an important analogy, drawn by others, between investment decisions and optimal timing of decisions to exercise call options. We thus linked software design decision-making to the theory of options. To the extent that the analogy is valid, we can expect options theory to provide insights into software design decision-making.

In particular, we believe that options concepts provide intellectual tools that can help us to better understand and to hone our software design decision-making heuristics. Ultimately, the theory of options, perhaps in conjunction with other advanced economic theories, might provide a firm foundation for what today remain informal and hard-to-grasp concepts such as "information hiding," "delaying of design decisions," "design for reuse," and so on.

To strengthen the case for the claim that options theory can help us to think about how to make design decisions, in this section we focus on how key parameters influencing options values affect corresponding software design decisions. In so doing, we seek to show software designers how concepts from options theory can be brought to bear in software design decision-making situations.

We do this in the context of the numerical example presented in the last section. In particular, we study how the value of the option to restructure depends on the uncertainty over the benefits of investing, the profitability of switching, and the probability of a favorable outcome, as well as on the direct cost of the design change L. We go on to show how the options approach reveals that the designer is not only an investor in concrete assets but in future opportunities. Options theory gives us a new view of the depth and complexity of the situation facing the savvy software designer.

Recall that the scenario where the technology succeeds was deemed favorable for restructuring because, in that scenario, the expected benefit S_1^f of restructuring at time 1 exceeds the direct cost L (expression 12). The scenario where the technology fails is unfavorable for restructuring because in that scenario the expected benefit S_1^u of switching at time 1 is less than L. In the following subsections, as we vary parameters, we continue to assume

$$S_1^f > L > S_1^u,$$

so that the two scenarios retain their favorable/unfavorable status.

9.1 Effect of Direct Cost

From expression 19, we notice that the value V_0 of the option is the maximum of two quantities: the NPV of strategy 1, namely $(S_0 - L)$, and the NPV of strategy 2, $(p/R)(S_1^f - L)$. Note that since $p/R < 1$, as L decreases, the former NPV increases faster than the latter. Thus, if all other parameters remain the same, there is a critical value for the direct cost L below which it is optimal to restructure immediately, i.e., at time 0.

Another way to state this principle is that if the direct cost L is sufficiently low, the cost of waiting (the profit $S_0 - L$ one would forgo) outweighs the value of waiting (the value V_0 of the flexibility to reverse the decision not to invest). Since there is nothing special about time 0, this statement applies at any time. Thus we provide a rigorous, options-theoretical justification for the following software design guideline:

If the cost to effect a software design decision is sufficiently low, then the benefit of investing to effect it immediately outweighs the benefit of waiting, so the decision should be effected immediately.

Although this design decision-making rule of thumb seems obvious, it contradicts a heuristic that one of the authors has heard promulgated on numerous occasions by recognized software experts, and which we cited at the beginning of this paper: Always delay making design decisions until you are forced to make them because they block progress on all other fronts. The plausible reasoning behind this rule is that you should wait until all possible information is in before investing. The options approach shows that this rule is wrong in general. Options theory also gives us a way to make precise the appealing notion that by reducing costs new technologies, such as restructuring tools [15], can toggle a situation from one in which it is best to delay to one in which immediate investment is optimal.

The options interpretation of software design decision-making teaches us that we are in the much more difficult position of having to strike a balance between the value of the benefits of investing immediately to have an asset now and the value of the flexibility lost when we make the irreversible decision to invest—and that that balance depends on a range of factors.

9.2 Effect of Uncertainty over Benefits B_n

In the numerical example of the previous section, the two possible values of B_n for $n \geq 1$ were $B_n^f = 2500$ in the favorable case and $B_n^u = -100$ in the unfavorable case. Now suppose we keep all parameters the same, except that we change B_n^f to 3000 and change B_n^u to -600. Notice in particular that the expectation of B_n,

$$\mathbf{E}(B_n) = 0.5 \times (3000 - 600) = 1200, \quad n \geq 1,$$

is the same as before, but that the *variance* of B_n is larger. This new parameterization models greater uncertainty about the rage of future benefits without any change in the net expected benefit, i.e., a “higher risk, higher return” project.

Since the expectation remains the same, the NPV of Strategy 1, (“restructure at time 0”), given by expression (15), is the same as before, because (see expression 9) the expected benefit S_0 of switching at time 0 depends only on the expectation of each B_n. On the other hand, if the software engineer waits for 1 month and switches only if the situation is favorable (Strategy 2), the net benefit S_k^f (see expression 12) is

$$S_k^f = B_1^f R/(R-1) = 3000 \times 1.1/0.1 = 33000, \quad k \geq 1,$$

which is bigger than the previous S_k^f value of 27500. Thus the NPV of Strategy 2 (see expression (16)) is bigger than before.

This shows that the incentive to delay the decision to invest in restructuring increases with project risk, manifested as uncertainty over future benefits B_n, as long as all else, notably the expected benefit, stays the same. Intuitively, this makes sense. The greater the uncertainty over the value of a manual—i.e., the greater the volatility in potential outcomes—the greater is the incentive to wait for better information before investing resources. With all else remaining the same, the value of options increases with the volatility of the value of the underlying asset.

To make the idea concrete, consider the options formulation of our design problem. The expected payoff of restructuring immediately is the same as before since the values $\mathbf{E}(B_n)$ are the same. However, if restructuring is delayed, then one of two outcomes occurs. In the unfavorable case (see (18)) the payoff V_1^u is still zero, because the design option will not be exercised. However, in the favorable case, the payoff V_1^f (see (17)) is greater than before. Thus the option value V_0 given by (19) increases.

In other words, the opportunity cost of restructuring immediately is greater, and all else remains the same, so there is more incentive to wait. Thus we can conclude with the following qualitative design guideline, which seems intuitively natural, but which we have now given a sound formal justification in terms of real options theory:

> *With other factors, including the NPV, remaining the same, the incentive to wait for better information before effecting a design decision increases with the uncertainty about (the volatility of) future benefits.*

Conversely, as uncertainty about the future value of a software asset diminishes, it becomes ever clearer whether or not it would pay to invest. In the limiting case of a certain future, one can decide immediately whether to invest or not based on the NPV. If a manual is extremely likely to be profitable, under our model there is little incentive to wait to write it. Similarly, if its value is clearly minimal or negative, a decision not to invest can be made immediately.

9.3 Effect of the Probability of a Favorable Outcome

In the example of the previous section, we assumed that at time 0 the likelihoods of favorable and unfavorable outcomes were equal, with $p = 0.5$. This probability distribution represents the risk that the favorable outcome will not be actualized. We now examine how the value V_0 of the real option depends on that probability p of a favorable outcome.

Consider the payoff $G_0 = (S_0 - L)$ from immediate exercise, i.e., the NPV of strategy 1 (see expression (14)):

$$G_0 = B_0 + (pS_1^f + (1-p)S_1^u)/R - L = (p/R)(S_1^f - S_1^u) + B_0 - S_1^u/R - L.$$

If we plot G_0 against p the slope would be $(S_1^f - S_1^u)/R$. The discounted expected value of the option V_1 is thus (expression 19)

$$\mathbf{E}V_1/R = (p/R)(S_1^f - L).$$

This is the NPV of strategy 2. If we plot this value against p, we find the slope to be $(S_1^f - L)/R$. Since we have assumed $S_1^u < L$, we see that as we increase p, the NPV of strategy 1 grows faster than that of strategy 2. Thus as the probability p of a favorable outcome increases, at some point the strategy of investing right away becomes optimal. To put it differently, as the risk of a unfavorable future decreases, so does the incentive to wait. We have thus given a rigorous basis for the following design decision-making heuristic:

> *The incentive to wait before investing varies with the likelihood of unfavorable future events occurring.*

Notice that this decision-making heuristic addresses uncertainty and risk in a different way than the previous rule. The previous rule addresses the variance in the payoffs under different outcomes. This rule addresses variation in the uncertainty about the likelihoods of future events that influence outcomes. We have thus identified two important and orthogonal dimensions of risk and have presented heuristics with rigorous theoretical underpinnings for reasoning about and responding to them.

9.4 Effect of Uncertainty over Direct Cost

In the example of the previous section we assumed that the direct cost L of restructuring is fixed and known at all times. We now examine the possibility of the cost L in the future being uncertain. This aspect of uncertainty is critical in software engineering, especially if delaying design decisions is an accepted strategy: It goes to the question of estimating project costs at future times. Uncertainty about costs might reflect uncertainty about availability of skilled labor in the future, or about changes in technology, such as the development of automated restructuring tools that could significantly reduce costs [15].

To simplify matters, let us assume that the monthly profit B_n from restructuring is 1500 at all times $n \geq 0$ (there is no uncertainty in this regard). Thus the expected benefit of switching at time k, for any $k \geq 0$, is given by an expression analogous to expression (12) (in either scenario):

$$S_k = B_k \frac{R}{R-1} = 1500(1.1)/(0.1) = 16500, \quad k \geq 0.$$

However, now assume that the direct cost L_0 at time 0 is known to be 10000, but that it is uncertain at time 1. Let us assume that L_1 is either $L_1^f = 5000$ (a favorable situation) or $L_1^u = 20000$ (an unfavorable situation). The NPV of strategy 1, investing now, is

$$NPV(1) = S_0 - L_0 = 16500 - 10000 = 6500,$$

which is positive. The traditional NPV rule suggests switching right away. Again, this rule is faulty because it ignores the contingent, option strategy: Wait a month, and switch only if the direct cost is $L_1^f = 5000$. The NPV of this strategy is

$$NPV(2) = (p/R)(S_1 - L_1^f) = (0.5/1.1)(16500 - 5000) = 11500,$$

which considerably greater than the NPV of the first strategy. Thus it is optimal to wait a month in this case before deciding whether to invest.

Now let us go a step further, and see what happens if we keep L_0 the same and increase the uncertainty (in particular, the variance) of L_1, while keeping its expectation $\mathbf{E}L_1$ the same. This would mean L_1^f is smaller, and $NPV(2)$ larger. In this case, the value of waiting is even greater. This situation is analogous to the one in Subsection 9.2. When the uncertainty over direct costs is larger, and the expectation remains the same, the potential profit in the favorable scenario increases, while in the unfavorable scenario it remains the same at 0. Thus we have provided a rigorous theoretical justification for another heuristic:

> *All else being equal, the value of the option to delay increases with variance in future costs.*

9.5 The Value of Information

In subsections 9.2 and 9.4 we showed how increasing uncertainty over the direct cost L and profit B_n increases the incentives to postpone the decision to commit the resources required to implement a design decision. However, this should not be taken to mean that uncertainty always leads one to delay all investments. Indeed, as we will see, quite the opposite is true.

For example, in situations where a small investment today produces information that dispels uncertainty about the actual but unknown state of the present world, it can be optimal to invest resources in prototyping experiments. As Boehm has argued [3], such investments are justified when the information that they reveal is worth more to the decision maker than its costs—e.g., when a small investment in a prototype averts a costly commitment to an unworkable design. More generally, information has value that can be quantified. This issue is important in decision analysis in systems engineering, and in decision-making with experimentation in particular. See Hillier and Liebermann for an introduction to and references on the topic [17].

The options view sheds additional light on the value of information approach—and on the valued of phased investments in particular. As we discuss in the next section, the options view also reveals an important, orthogonal dimension in which early investments under uncertainty can be justified: not only under uncertainty about the present state of nature, but also about how the future will turn out (e.g., whether certain markets will evolve in favorable ways). We will illustrate both of these ideas, in this subsection and the next, in the context of our design restructuring example.

To begin with, suppose that the software engineer can perform the restructuring for information hiding in two phases. The first phase costs 1000. With probability 0.5, that expenditure will be adequate to restructure the system; but with probability 0.5 another 3000 dollars will have to be invested to produce a satisfactory new design. It might be that the selected restructuring tool turns out to have some unforeseen shortcomings that require some manual restructuring, at a much greater cost. More generally, the project might face what have been called technical risks—we will call them internal risks—that can be resolved only by investigating the actual state of nature (to use Boehm's phrase). In our case, investing in the first phase gives the designer valuable information about the costs and benefits of further investments.

Let us suppose that, once completed, the profit from restructuring at each time t is 200. Thus the present value of the profit stream at any time is $200R/(R-1) = 2200$. The traditional investment analysis compares the expected cost of restructuring, $1000 + (0.5)(3000) = 2500$, with the present value of profits, which is 2200. Owing to the risk that nature is in an unfavorable state (that the restructuring tool actually has as yet unknown shortcomings) the NPV is negative, and it would seem that one should not invest in the first phase of the restructuring.

However, this analysis ignores the value of the information obtained from completing the first stage of the implementation, and the fact that the engineer can abandon the project if a second phase costing 3000 turns out to be necessary. That is, the NPV analysis ignores the contingent strategy—and thus the designer's good judgement in the face of information revealed by the first phase. In particular, in this case, the traditional analysis ignores the value of the option that the manager has to cancel or continue the project in light of the information revealed by the first phase. The corrected NPV is $(0.5)(2200) - 1000 = 100$.

In other words, for 1000 today the designer can buy an asset that with even odds is worth either 2200 or 0 and find out whether or not it's a winner. It's clearly a good bet: 1000 for 1100 in expected value. If the bet turns out to be a loser, the designer, qua strategist, leaves the table, refusing to commit 3000 more to a losing proposition, but knowing that she spent her 1000 well, despite what might be seen as the "failure" of the project.

The designer as strategist thus expects to lose some bets, even good ones. The designer who "anticipates" incorrectly, for example, has not necessarily made a design error. You're not a loser because you lose some bets; you're a loser if you make losing bets. In this sense there's real wisdom is the saying that, *it's not whether you win or lose but how you play the game.* The designer should invest in the first phase of the project despite even odds of an unfavorable outcome because the expected value of the option outweighs its cost. The options view thus highlights the strategic dimension of software design. In this example, the designer plays a strategy game against the present but uncertain state of nature. We thus derive another heuristic rule, which has a rigorous justification in options theory:

> *If investing a little today reveals information about a state of nature that determines whether a follow-on investment is wise, and if the cost of that information is low in relation to the potential value of the follow-on investment, then it is a good idea to make the initial investment, even if it ultimately shows the follow-on investment to be unwarranted.*

From this insight comes an additional rule for managers.

> *Don't punish designers for unfavorable outcomes, but only for unwise investments. Reward designers who take intelligent risks.*

We view the right to make a decision about a follow-on investment in light of information revealed by an earlier investment as an option. In our example, exercising the initial option to invest 1000 yielded a second option in the form of the right to discontinue the project. In general, exercising one option could buy another, and so on, for many steps. The initial 1000 buys an option to discontinue the project to develop the tool. As we will see in the next section, the tool might itself embed an option to restructure the system. Restructuring in turn might provide the option to adapt to an increase in demand should that event actually occur. The actual profitable opportunity might be many steps away.

This insight is of fundamental importance in software design decision-making. Traditional software economics views overlook the strategic value of the options—i.e., flexibility—embedded in development projects and assets. The options approach is not only superior to traditional NPV analysis for quantitative reasoning about projects and assets with embedded options, but even more importantly for our purposes in this paper, it provides a firm theoretical foundation for understanding and characterizing the critical value of good strategy in design. The ability to make investment decisions contingent on the resolution of internal and external, present and future uncertainties is really of the essence in software design.

9.6 The Value of Opportunity

Just as the options view shows that it can make sense to invest in the face of uncertainty about the present state of nature, so it shows too that it can pay to invest in the face of uncertainty about the future. Again the key is that an early investment can create an option for follow-on investments. If there's a chance that the future will turn out favorably, then it can be strategically wise to invest some today to “keep your foot in the door.” Options theory shows that it can be wise to invest today even if the expected value of both the initial and of the follow-on investments are negative according to a traditional NPV analysis.

If the option to continue investing is worth more than it costs, then it makes sense to invest in it, even if a traditional NPV analysis shows the investment to be unwarranted. The traditional analysis overlooks the value of having the flexibility to respond to changing conditions in the future. Furthermore, it overlooks the value of flexibility in real assets that are already held, e.g., the value of flexible architectures, reusable assets, software that has been designed for change, and software that provides the ability to take advantage of opportunities that might arise. Again the options view can provide significant insights by giving us a sound basis for reasoning about the value of flexibility, or the lack thereof, in these dimensions. We thus frame the following guideline.

> *As uncertainty grows, consider making investments today in design assets that might appear to be unjustified on traditional software engineering economics grounds but that have potential for significant payoffs should conditions turn favorable.*

For project managers with an analytical bent, the following fact is of course critical:

You can analyze the value of flexibility embodied in such assets using options pricing models, given specified assumptions about cash flows, probabilities of events or states of nature, variance in costs and benefits, interest rates, the time period within which you have the flexibility to invest.

This is not a line of reasoning that we wish to pursue in this paper. However, we are confident that as people come to appreciate the value of an options view of software design—in reuse, iterative and spiral development processes, legacy systems, and software architecture, for example—that options theory will emerge as an important quantitative tool for software management. The mathematical foundation for rigorous valuation of flexibility in the form of options was laid by Black and Scholes in 1973 [2] and remains useful today for such purposes.

To make these points more concrete (but with the arithmetic elided), let us return to our fictional restructuring problem. Consider the following alteration in the situation. Let's assume that the project to develop the new agent technology faces certain technical barriers. Assume, too, that if it overcomes them, then there's a good chance that the demand for agents will soar. In this case, the ability to restructure quickly will have a much greater likelihood of being valuable. However, if the barriers are not overcome, the likelihood of a significant increase in demand for agents will remain low. In that case, investing in restructuring will continue to have a poor expected value.

Suppose, furthermore, that if the agent technology succeeds—let's say one year from now—then we will have a window of opportunity that will close three months thereafter. Restructuring within three months so as to accommodate the growth in demand for agents will position our product to succeed. However, if we're unable to restructure within that time, then the competition will beat us to market with a product that accommodates the change effectively. In that case, they'll win, locking us out of the market, and closing off the opportunity for us to restructure our system to capture the potential profit.

Finally suppose that today we do not have the ability to restructure our product within three months, owing to lack of experience and the necessary infrastructure to use our restructuring tools effectively. Perhaps we plan to use Refine [21] as a tool, which requires a custom front end that does not yet exist, and that will take more than three months to build.

Given the uncertainty about whether our system will actually have to accommodate rapid change—a factor that is contingent on the success of the agent development technology, and still somewhat uncertain even in the case of success—should we commit resources today to a restructuring project? A more modest question is, Should we invest a little now to develop a restructuring tool that might become valuable in the future?

The standard approach says commit to either tool development or to restructuring if the traditionally computed NPV of either project is positive. If the NPVs of both tool development and restructuring are negative under such an analysis, does that mean there's no incentive to invest? The options approach shows that there can still be such an incentive.

Clearly, a simple options-based strategy of delaying investment until the key future events are known is unattractive, because if we don't invest anything today, and then if the agent technology does succeed, we won't have time to take advantage of the opportunity, because we won't be able to restructure our system quickly enough.

Fortunately, there is another strategy: We can invest a little today to build a restructuring tool that positions us to exploit growth in the agent market, should it occur. Although the tool might have some intrinsic value of its own, that value might not outweigh its costs. But the real value of the tool includes the opportunity to restructure quickly, which in the scenario we hypothesize amounts to an ability to exploit a potentially lucrative opportunity. Realistic scenarios can be constructed easily in which the NPV of both the tool and restructuring are negative under a traditional analysis but in which an options-based analysis shows investing in the tool to be a good bet. The value of the ability to exploit a potentially lucrative opportunity can more than compensate for the apparently negative value of the tool. The value of the flexibility afforded by the tool, overlooked by traditional software engineering economics approaches, is properly understood and accounted for as a real option.

The critical point throughout this paper is that options have value because they represent opportunities in the form of the flexibility to make contingent investment decisions. The savvy designer is thus aware that she's responsible not only for exercising options, but for investing strategically to create options—as in our hypothetical restructuring tool. Moreover, options theory explains how incentives to create options vary with uncertainty and with other factors. Real options theory provides a rigorous basis for modeling the value of such strategic investments.

Readers interested in a deeper but elementary discussion of the value of expansion flexibility, in the options interpretation of flexibility, and in a worked example are referred to Chapter 21 of Brealey and Meyers, on applications of options pricing theory [5].

9.7 On Design as an Anticipatory Activity

Let us now reconsider in light of the preceding discussion the old idea that software design is an anticipatory activity. Put simply, the idea is that if you guess right about what's likely to change and then design accordingly, “you win,” otherwise, “you lose.” By appealing to the well-developed theory of options, we have shown this idea to be simplistic.

Rather, viewing design decisions as real options leads us to see the designer as a manager of irreversible capital investment decisions in the face of uncertainty. As such, she has to play a much more subtle and interesting game. She not only decides whether or not to invest today, nor only when if ever to invest in real assets. She also has to think about how and when to invest to create opportunities to make additional potentially lucrative investments. She views not only artifacts but also flexibility and opportunity as assets having tangible value.

Returning to the issue of information hiding, we can see that the savvy designer's thinking might go beyond designing interfaces to hide design decisions that are deemed “likely to change.” She might for example design interfaces to hide decisions that are deemed “not very likely to change,” but where the payoff in the unlikely but favorable scenario is large. More generally, the architect is is a position of having to make strategic investments in flexibility.

She has to split investments between cash-flow-producing investments (e.g., programs that work today) and those previously intangible aspects of design that we can now interpret as options (e.g., the flexibility to adapt to changing requirements). Moreover, she is in an iterated “game,” in which she has to make such decisions adaptively as time passes and as various uncertainties are resolved.

Like any capital investment manager, the designer gets paid to take strategically sensible risks. The options approach helps us to see the real complexity of software design decision-making. Design is an anticipatory activity, but it is not simple or straightforward. That view of design sees the designed artifact as a passive investment and does not account for the value of flexibility to respond to emerging contingencies. By contrast, we view design not just as anticipatory but as a *strategic activity* in which one tries to make sequences of good bets on future and other uncertain outcomes.

To the extent that a good design is a flexible design (reflecting Parnas's dictum, “design for change”) we thus have a rigorous basis in options theory for placing a value on good design above and beyond the value of merely having a program that solves today's problem or even today's and those of tomorrow that we know today will occur (such as the coming of Year 2000). In our view, the value of a good design is also in the value of options embedded in the design. Conversely, we can define the legacy system as one that might have enormous use value but little value in the form of embedded options—i.e., that is has little flexibility.

To the extent that options theory gives us a way to place a value on flexibility, it might even give us a concrete way to convince even skeptics of the value of good software design. In principle, perhaps even in practice, options theory gives us a way to put a quantified value on flexibility—and to evaluate whether a given investment in flexibility is a good bet. We think that good designers and managers already act in ways consistent with the insights that we have formalized by appealing to real options theory.

10 Conclusion

We started this paper with the claim that current software design doctrine, cast in such terms as information hiding, delaying design decision, and software reuse, is hard to understand and to justify, and that one key reason for this difficulty is that current design doctrines lack sound or adequate theoretical foundations. We have taken an economics stance on the issue of foundations, and have, in particular, advocated a real-options-based approach to evaluating, improving, and generating software design decision-making heuristics.

We presented the analogy between capital investment decisions and financial call options, which has gained considerable prominence in recent economics research. We then contributed the insight that there is a strong analogy between decision-making in software design and in capital investing. We thus justified our appeal to real options for a theoretical foundation for software design decision-making guidelines.

Next, we presented the basic concepts of options theory, with enough mathematical background for a person to understand options valuation at a basic level. We also presented a simple worked example to show how the concepts can be applied to a simple, but still surprisingly subtle, decision problem in software design. The decision problem we selected was about whether or not to restructure a software system to impose an information hiding interface that would ease a set of changes that might or might not be needed.

Next we showed how options values change as critical parameters are varied, and how those behaviors can lead us to well founded design heuristics. We ended up deriving, with rigorous justification, a set of software design decision-making heuristics. For example, it can pay to invest some now, even in areas that do not at present appear profitable, to create potentially profitable opportunities. From an architectural point of view, this suggests that it might sometimes be profitable to build flexibility in a system not in anticipation that given changes are likely but because the payoff in the unlikely event that change is needed is high enough relative to the cost to justify the investment.

Of course, being able to value flexibility can also give the manager a well founded justification for controlling the flexibility-obsessed designer. There is a price above which an option is not a good buy. Flexibility is neither good or bad; it's just a worthwhile investment or not. Moreover, whether it is or isn't depends on a set of factors that might not be obvious to the causal software designer.

The maximization of value, which is the economic objective of any serious enterprise, requires that everyone be a savvy investor of the firm's capital. Software designers have an essential role to play because they control vast investments in real software assets. They can perhaps do better if they understand that the informal heuristic concepts they employ, such as “design for change,” can perhaps be given sound theoretical foundations, and that those foundations can be used to evaluate heuristics and to reason about their “operational ranges,” i.e., about when they do and don't make sense.

Options theory also gave us insights into the strategic value of phased approaches such Basili and Turner's iterative development and Boehm's spiral model. Each iteration gives us information and the opportunity to make strategic decisions at the next step, including the decision to cancel the project, or to delay pending resolution of certain unknowns. Earlier investments yield both productive assets and options, both of which we can now value rigorously.

Finally, the options view made precise the value of investing under uncertainty not only in prototypes, which resolve uncertainties about the present state of nature, but also in what we'll call “wedges:” products that are not currently attractive but that embody options to exploit lucrative opportunities that might emerge in the future. A wedge is a valuable “foot in the door” in the form of an option to invest should the future be favorable.

We are obviously not the first to notice that flexibility is of the essence in software design, and that flexibility has both costs and benefits. Parnas's work on information hiding [26], ease of extension and contraction [27] and families of systems [25] goes to the heart of the matter, albeit without explicit appeal to financial concepts. More recently, Fayad and Cline [12], to cite just one example, emphasize that design flexibility has costs, that flexibility should be designed in in those areas where it makes the most economic sense. They identify *design patterns* [14] as providing the “hinges” that are needed for flexibility in particular dimensions. They even state that such hinges provide “opportunities” to make future changes that might be necessary—using a word that clearly reflects a real options mode of thinking.

Nor, obviously, are we the first the employ economics-based approaches to reasoning about software design. We have cited several seminal and important works in that area. Much work has been done in recent years in laying economic foundations for analyzing investments in software reuse, for example.

However, to the best of our knowledge, we were the first to identify real options explicitly as a rigorous theoretical foundation for a wide variety of important software design concepts [32]. Shortly after our early paper, Withey published a report [35] in which he presents what is essentially a real-options approach to analyzing investments in reusable assets for software product lines. By contrast to our work, his is not meant to present the underlying mathematics rigorously. Also, Withey seeks analysis techniques intended to evaluate specific reuse projects. That is an important direction for research with obvious and important practical uses. In this paper, however, we focus on how the rigorous theory of options can help us to explain, improve and generate qualitative software design guidelines. It is important to do so because in practice, software designers rely so heavily on such guidelines. Nevertheless, the existence of an underlying theoretical foundation is obviously beneficial in that it providess recourse to those who seek to perform quantitative analyses in specific project contexts.

Before concluding, we emphasize again that we are not proposing a silver bullet. The optimal decision in any given case obviously has to be evaluated based on estimates of the relevant parameters. The relevant factors include when the flexibility would be exploited, the likelihoods of various outcomes, the interest rate, and the costs and benefits and their respective variances. Such parameters can be hard to estimate for real projects. Furthermore, we have barely scratched the surface of modern options theory. It is known that some problems in options valuation are computationally intractable. Nor do we want to suggest that options theory as we have presented it is the only way to interpret software design decision-making effectively.

A conceptual unification is needed, and we believe that we have shown that options concepts can help to reveal commonalities underlying a range of important software design concepts. Our goal was to show that there is at least one well developed theory and body of knowledge that we can bring to bear to improve the discourse on software design, and to improve software design heuristics. We believe we have made progress in that dimension. However, we also see a potential danger in unjustified borrowing of subtle ideas from other fields. Applications of options thinking to software should of course be done with care and discrimination.

To conclude, we have made progress toward a theory in terms of which we can justify and refine a wide variety of important, widely used software design decision-making heuristics and concepts. We believe that a variety of useful concepts and practices can be analyzed and interpreted in terms of options theory and other economic theories. However, given the uncertainties involved in introducing new ideas into an established field, we feel it prudent to delay investing additional resources until we have more confirmatory evidence of the value of the basic approach.

Acknowledgements. This work was supported in part by the National Science Foundation (NSF) under grant numbers CCR-9502029 and CCR-9506779, and by the Defense Advanced Research Projects Agency and Rome Air Labs under grant number F30603-96-1-0314.

References

[1] V. R. Basili and A. J. Turner. Iterative enhancement: A practical technique for software development. *IEEE Transactions on Software Engineering*, SE-1(4):390–396, Dec. 1975.

[2] F. Black and M. Scholes. The pricing of options and corporate liabilities. *J. Political Economy*, 81:637–654, 1973.

[3] B. W. Boehm. Software engineering economics. *IEEE Transactions on Software Engineering (SE)*, 10(1), Jan. 1984. Also published in/as: Prentice-Hall publishers, 1981.

[4] B. W. Boehm. A spiral model of software development and enhancement. *IEEE Computer*, 21(5):61–73, May 1988.

[5] R. A. Brealey and S. C. Myers. *Principles of Corporate Finance*. McGraw-Hill, 5th edition, 1996.

[6] T. H. Cormen, C. E. Leiserson, and R. L. Rivest. *The Design and Analysis of Computer Algorithms*. Cambridge, Mass.: MIT Press, 1990.

[7] J. Cox, S. Ross, and M. Rubinstein. Option pricing: A simplified approach. *J. Financial Economics*, 7:229–264, 1979.

[8] G. Daily. Beyond puts and calls: Option pricing as a powerful tool in decision-making. *The Actuary*, 30(3), 1996.

[9] A. Dixit and R. Pindyck. *Investment under uncertainty*. Princeton Univ. Press, 1994.

[10] A. Dixit and R. Pindyck. The options approach to capital invesment. *Harvard Business Review*, pages 105–115, May-June 1995.

[11] J. Favaro. A comparison of approaches to reuse investment analysis. In *Proceedings Fourth International Conference on Software Reuse*. IEEE, Apr. 1996.

[12] M. Fayad and M. P. Cline. Aspects of software adaptability. *Communications of the ACM*, 39(10):58–59, Oct. 1996.

[13] J. Flatto. *The application of real options to the information technology valuation process: A benchmark study*. PhD thesis, Univ. of New Haven, 1996.

[14] J. Gamma, Helm and Vlissides. *Design Patters: Elements of Reusable Object Oriented Software*. Addison Wesley, NY, 1994.

[15] W. G. Griswold and D. Notkin. Automated assistance for program restructuring. *ACM Transactions on Software Engineering and Methodology*, 2(3):228–269, jul 1993.

[16] A. Habermann, L. Flon, and L. Cooprider. Modularization and hierarchy in a family of operating systems. *Comm. of the ACM*, 19(5):266–272, May 1976.

[17] F. S. Hillier and G. J. Liebermann. *Introduction to Operations Research*. McGraw Hill, 1995.

[18] J. Hull. *Options, Futures, and Other Derivative Securities*. Prentice Hall, 2nd edition, 1993.

[19] C. W. Kreuger. Software reuse. *ACM Computing Surveys*, 24(2):131–183, June 1992.

[20] P. Maes. Agents that reduce work and information overload. *Comm. of the ACM*, 37(7), July 1994.

[21] G. K. . L. Markosian. Applications of refine language tools to software quality assurance. In *The Ninth Knowledge-Based Software Engineering Conference*, pages 20–23. IEEE Computer Society Press, 1994.

[22] R. McDonald and D. Siegel. The value of waiting to invest. *Quarterly Journal of Economics*, 101:707–727, 1986.

[23] R. C. Merton. *Continuous-Time Finance*. Blackwell, 1990.

[24] W. J. Mitchell. *Space, Place and the Infobahn: City of Bits*. MIT Press, 1996.

[25] D. Parnas. On the design and development of program families. *Transactions on Software Engineering*, SE-2, Mar 1976.

[26] D. L. Parnas. On the criteria to be used in decomposing systems into modules. *Communications of the Association of Computing Machinery*, 15(12):1053–1058, Dec. 1972.

[27] D. L. Parnas. Designing software for ease of extension and contraction. In *Proceedings of the Third International Conference on Software Engineering*, pages 264–277. IEEE, 1978.

[28] D. L. Parnas, P. C. Clements, and D. M. Weiss. The modular structure of complex systems. In *Proceedings of the* 7^{th} *International Conference on Software Engineering*, pages 408–419. IEEE Computer Society Press, Mar. 1984. This paper was cited as the best/most influential paper by the program committee for ICSE 17 in 1994.

[29] E. Rechtin and M. W. Maier. *The Art of Systems Architecting*. CRC Press, Boca Raton, Florida, 1997.

[30] M. Shaw and D. Garlan. *Software Architecture: Perspectives on an Emerging Discipline*. Prentice-Hall, 1996.

[31] Sick. Real options. In *Handbook of Finance*. Elsevier/North-Holland, 1995.

[32] K. Sullivan. Software design: the options approach. In *Proc. Software Architectures Workshop*, Oct. 1996.

[33] K. Sycara, K. Decker, A. Pannu, and M. Williamson. Distributed intelligent agents. Technical report, Carnegie Mellon University, 1996.

[34] L. Trigeorgis. *Real Options*. MIT Press, 1996.

[35] J. Withey. Investment analysis of software assets for product lines. Technical Report CMU/SEI-96-TR-010, Carnegie Mellon University/Software Engineering Institute, Nov. 1996.

[36] W. Wulf. *Personal Communication with Kevin Sullivan.*