Software Design: The Options Approach

Kevin J. Sullivan
University of Virginia Department of Computer Science
Thornton Hall
Charlottesville, VA 22903
(804) 982-2206
sullivan@Virginia.edu

Abstract

Many software engineering principles and concepts that are critical to reasoning about problems in software design (of which software architecture is an important special case) remain ad hoc, idiosyncratic and poorly integrated. I argue that this is due to our lack of a clean theory about how to make software design decisions. In this paper I propose that we should view software design as a process of deciding how to make irreversible capital investment in software assets of uncertain value, and that financial options theory provides a firm, unifying, simplifying and well developed basis for such decision-making. To support this view, I interpret software architecture and other related concepts in options theoretic terms.

1 Introduction

Many important advances have been made in the theory and principles of software development over the past decades. Nevertheless, many of our most important principles and explanations remain ad hoc, idiosyncratic, and poorly integrated. Try to explain, for example, the connections among the concepts of risk, likelihood of change, information hiding, architecture, program families, and resource allocation. The connections are deep, but subtle and hard to understand. Is is that our subject is inherently opaque?

I don't think so. Rather, our difficulties are artifacts of an inadequate theory. We haven't yet figured out how to look at what we see. Much as epicycles were needed to explain planetary motion before the simplifying heliocentric theory, our confusion reflects the need for a clean theory about how to invest in software assets.

I propose that we can clear up some of these difficulties by looking at software engineering as deciding how to make irreversible capital investments in the face of uncertainty. We get real leverage when we take the second step of reasoning about such capital investment decisions in terms of financial call options [4]. We can then reason about software design in terms of call options, for which there is a well-developed theory and body of knowledge.

This work was supported in part by the National Science Foundation under grants CCR-9502029 and CCR-9506779.

What is a call option? A call option confers, for a period of time, the right but not the obligation to purchase, at a given price, an asset whose future value might be unknown. Exercising an option trades the price for the asset. The trade is irreversible in that you do not have the right to go back on the trade, and the option itself is nullified. To hold an option, by contrast, preserves the right to exercise the option in the future until such time as the option expires. To hold an option is a reversible decision. When the future value of the underlying asset is uncertain it might therefore be best to hold an option—to delay investing until more is known. On the other hand, holding an option forgoes the benefits of owing the asset today (e.g., revenues that it might generate), and thus can have a significant opportunity cost. The essence of good capital investing under uncertainty and irreversibility is thus in creating and valuing options and in optimally timing decisions to exercise them (or not). Options theory provides a basis for such decision-making [4].

In this paper I make and present evidence for three claims. First, many software design decisions amount to decisions about capital investment under uncertainty and irreversibility. Second, we can understand important software design principles and concepts as ad hoc, idiosyncratic rules that implicitly reflect the capital investment character of software design and that implicitly embody options-based strategies. Third, we can simplify, unify, rationalize and even improve important software design concepts by placing them explicitly on an options theoretic foundation.

2 Aphorisms

To get a sense of options thinking in software design, consider the following aphorisms and options-based interpretations.

1. A friend once told me his managers were so uncertain of what they wanted and changed their minds so often that he had to design a system that was "all hooks." A student described another experience: His company president invested so much in an infrastructure to make C++ look like Smalltalk that the product never got built.

   In the first case, management's demand for a system established that some system would be valuable. However, their uncertainty seemed to justify investing in a portfolio of options—the hooks needed to build system variants at low cost. But because only one system was needed, most of the investment in options was lost. More seriously, the options could not just be written off, because their architectural manifestation as unnecessary complexity in the form of hooks had ongoing carrying costs. Thus what were assets in the form of options became liabilities that could not easily be scrapped. In the second case, the president

invested so much to create non-revenue-producing options to build systems using Smalltalk idioms that he ran out of capital to invest in revenue-generating assets. Software designers have to balance investments in creating, valuing and exercising options.

2. Good software engineers understand the value of delaying design decisions. Procrastinating preserves flexibility and may avert investments in worthless assets.

   Delaying commitments is widely understood to be a key software engineering principle [7]. Options theory helps to explain why. Design decisions are like call options. To bind and implement a design decision is to exercise an option—to invest in a software asset. Exercising an option is an irreversible act; but delaying, like holding an option, is not. If asset values are uncertain, it can pay to delay; but delaying also forgoes the benefit of having the asset now. You have to weigh the value of investing now against the value of investing at all possible future times [9]. Timing irreversible investments in software assets of uncertain value is of the essence.

3. Software methodologies often posit *rigid dictums*. Write a specification before design. Write a user's manual. Design for change. Get the architecture right first. Always use information hiding. Don't make decisions until not doing so blocks all progress.

   Rigid dictums are overly simplistic investing rules of thumb. It's usually possible to construct realistic scenarios in which they lose. It is never free to create documents or architectures or to design for change. Whether or not investing at a given time is optimal is not a simple question. Options theory is concerned with answering such questions. Rigid dictums are unreliable proxies for optimal investment strategies.

3 Investing

In more detail, I connect options theory to software design in two steps. First, we can understand software design as the creation, valuation and exploitation of opportunities to make irreversible capital investments in software assets of uncertain (present and future) value. Architectures, documents, program generators, and information hiding interfaces are examples of assets. Second, we can rationalize decisions about whether and when to make such investment by looking at investment opportunities as call options [4] and by appealing to options theory (e.g., [9], see also [8]).

Let's start with the idea that investment opportunities are call options. The analogy is as follows. Managers decide how to exploit opportunities to invest capital in such things as plant or equipment. The future value of such an asset is uncertain and may depend on such factors as future demand. Moreover, capital investments of this kind are frequently irreversible. Once you buy a capital asset, such as a nuclear power plant or a manufacturing system, you're stuck with it. To be relieved of the asset requires that you pay to scrap it. Scrapping itself is viewed as an additional capital investment the return on which is a (possibly uncertain) reduction in future costs [4].

The isomorphism between call options and capital investments invites the application of the theory of call options to capital investment decision-making. Mathematical options theory is beyond the scope of this paper. However, I do highlight eight options theory concepts that are particularly important and relevant here.

1. Options let you wait to resolve uncertainties before deciding whether to invest.

2. Options can be created through capital investments.

3. Options themselves can have considerable value.

4. The value of an option increases with the level of uncertainty about the value of the underlying asset.

5. Uncertainty creates incentives to create and hold options.

6. Not to exercise an option can incur considerable opportunity costs.

7. The cost to exercise an option includes the cost of the investment plus the lost value of the option itself. The lost value of the option has to be accounted for.

8. An asset's value includes both the revenues it produces and the value of options it embodies or creates.

Now for the connection to software: We interpret software design as the creation, valuation, holding and exercise of options to make irreversible investments in software assets of uncertain value. These investments have the characteristics that makes options theory applicable. First, the asset values are often uncertain. The present value of a software architecture depends on future requirements changes, for example. Second, such investments are generally irreversible. The development costs are sunk and the assets have their full value only in a given project, domain, or development group.

We can thus look at the ad hoc, idiosyncratic principles of software design from the perspective of options theory. Aren't many software design guidelines really just implicit investment advice? If so, can we understand and improve existing principles and guidelines or even create useful new ones by appealing to options theory?

4 Evidence

I can't answer these questions definitively in this paper. What I do in the next section is to present some evidence to justify exploring the idea further. The evidence is in the form of reinterpretation in options theoretic terms of software design principles in three areas. The areas are software architecture [5], generators [1], and the spiral software process model [2].

4.1 Architecture

An architecture is not a revenue-producing asset—an application. Rather, it embodies a portfolio of options to invest small additional amounts to create such assets. To design an architecture is to make a capital investment to obtain such a portfolio of options. Options theory gives us a way to think about software architecture as distinct from software applications.

The options view fosters additional insights. We can also use it to distinguish legacy from non-legacy applica-

tions. Generally, applications not only produce revenue but provide additional value in the form of options, insofar as they can be evolved at relatively low cost to meet future demands. Isn't a legacy system, then, just one that produces (perhaps considerable) revenue but that has little additional value in the form of options?

We can also interpret information hiding [10][11] in options terms. To employ information hiding is to decide to invest capital today in an information hiding interface to obtain the right to change the hidden secret of a module at low cost in the future. An information hiding interface creates an option to make certain changes. Such options are not free. Should one invest in information hiding interfaces, and if so, when?

Options theory suggests that it won't always be optimal to invest in interfaces early. The value of the option to change the secret, obtained though the investment in the interface, has to outweigh not only the capital cost of the interface itself but also the value of the option to delay creating the interface. The incentive to delay investing in interfaces is increased by the difficulty of scrapping such investments. Badly conceived interfaces amount to unwanted capital with high carrying costs. I often delay defining interfaces because of uncertainty about how implementation elements relate. Perhaps we can now better explain why delaying investments in information hiding interfaces sometimes make sense.

Finally, we can interpret restructuring of software systems [6] as capital investments that are made to scrap assets obtained through prior investments (e.g., obsolete or suboptimal interfaces). The return on the investment in scrapping is a reduction in future costs (c.f., [4]). Options theory provides a framework for thinking about restructuring, and perhaps can help us to evaluate and time decisions to scrap earlier design commitments.

4.2 Generators

Next, like architectures, we can understand domain-specific program generators and the reference architectures they employ [1] as portfolios of options. They are not revenue-producing applications but tools that provide options to purchase members of a family of systems for the relatively small cost of easy-to-write specifications. The value of a generator is in both the value of the application-building options it provides as well as in options to enhance the generator. The cost to create a generator includes both the capital investment per se as well as the value of the forgone option to delay investing in the generator. It appears that options concepts give us a way to reason about the value of investments in such software assets as program generators.

4.3 Spiral Model

The spiral model is risk driven. Risk means uncertainty. Uncertainty puts a premium on options, increasing the incentives both to delay investments and to invest in more options. The spiral model can be explained in options terms. Developing alternatives in the spiral model amounts to investing to create options. Not always writing specifications before implementation (for example) amounts to holding rather than exercising options. Risk assessments are capital investments made to create options and to estimate their values more accurately. We thus interpret the spiral model as an optimizing approach to investing in the creation, valuation and exercise of options to develop software assets. Beyond such qualitative interpretations, options theory might help answer questions such as how much to invest in risk assessment and when.

5 Related Work

One thing I have not done in this paper is to discuss software architecture in typical systems theory terms—of abstract models formulated as entities and relations (or components and connectors[5]); of isomorphisms, homomorphisms, differences or other sorts of correspondences between different abstract models or between models and reality; or in terms of taxonomies of abstract models. The systems theory perspective is indispensable. My intent is not to ignore it but to describe an orthogonal, novel and valuable view: of architecture as a risky asset, and of software design more generally as, in important ways, the wielding of call options. At the center of concern, then, is not so much structure as value—value in an uncertain world in which there are conflicting incentives to make and delay investment decisions that cannot be unmade.

The view of software artifacts as capital assets is obviously not new, nor of software development as capital investment. DeMarco has written, "System architecture is expensive, but probably not as expensive as its absence [3]." On the other hand, viewing software as capital is not enough, and such views can leave us with just more rigid dictums: e.g., invest in architecture. In this paper, I suggest that we begin to *recognize* such dictums, that we *explain* the wisdom they embody in options theoretic terms, and that we *improve* them by crafting explicit qualifications and justifications on the basis of insights provided by options theory. To the best of my knowledge this options view of software design is novel.

6 Summary and Conclusion

Software design concepts and principles today remain ad hoc, idiosyncratic and not well integrated. Despite much research, software design concepts remain hard for many people to understand, practice, teach, and learn. Like the epicycle explanation of planetary motion, they work, but are conceptually and probably economically suboptimal.

I have argued that we can resolve some of our difficulties by viewing software design as deciding how to make irreversible capital investment in risky software assets, and by appealing to options theory for insights and a basis for explanation and decision-making. Early evidence appears to support this view. In particular, the options perspective appears to give insight into what otherwise remain hard-to-understand concepts, such as the value of delaying design decisions, the difference between legacy and non-legacy systems, and the advisability of information hiding in the face of likely change.

The feasibility of the profitable use of quantitative analysis based on mathematical options theory to aid in software design is unclear and is an issue to be addressed in future work.

7 Acknowledgments

David Notkin emphasized the need to explain why rigid dictums are often suboptimal. Ram Kumar emphasized how opportunity costs counterbalance incentive to delay investing. Mark Marchukov provided the anecdote about the Smalltalk layer on C++. Andy Litman told the one about the system that was all hooks. This work was supported in part by the National Science Foundation under grant numbers CCR-9502029 and CCR-9506779.

References

1. Batory, D., L. Coglianese, M. Goodwin and S. Shafer, "Creating reference architectures: an example from avionics," Proceedings of SSR'95, *Software Engineering Notes*, April 28-30, 1995, pp. 27-37.

2. Boehm, B.W., "A spiral model of software development and enhancement," *IEEE Computer* (21,5), May, 1988, pp. 61-72.

3. DeMarco, "On systems architecture," The Atlantic Systems Guild, September 12, 1995, pp. 26-32.

4. Dixit, A.K. and R.S. Pindyck, "The options approach to capital investment," *Harvard Business Review*, May-June, 1995, pp. 105-115.

5. Garlan, D. and M. Shaw, "An introduction to software architecture," *Advances in Software Engineering and Knowledge Engineering*, Vol. 1, World Scientific Publishing, 1993.

6. Griswold, W.G. and D. Notkin, "Automated assistance for program restructuring," *ACM Transactions on Software Engineering and Methodology* (2,3), July, 1993, pp. 228-269.

7. Habermann, A.N., L. Flon and L. Cooprider, "Modularization and hierarchy in a family of operating systems," *Communications of the ACM* (19,5), May, 1976, pp. 266-272.

8. Kumar, R., "An options view of investments in expansion-flexible manufacturing systems," *International Journal of Production Economics 38*, 1995, pp. 281-291.

9. McDonald, R. and D. Siegel, "The value of waiting to invest," *Quarterly Journal of Economics* (CI, 4), November, 1986, pp. 707-727.

10. Parnas, D., "On the criteria to be used in decomposing systems into modules," *Communications of the ACM*, December, 1972, pp. 1053-1058.

11. Parnas, D., "Designing software for ease of extension and contraction," *IEEE Transactions on Software Engineering*, (SE-5,2), March 1979, pp. 128-138.

12. Shaw, M. and D. Garlan, *Software Architecture*, Prentice-Hall, 1996.