

Opportunity-Centered Software Development

Kevin Sullivan
University of Virginia Department of Computer Science
151 Engineer's Way
Charlottesville, VA 22904 USA
sullivan@virginia.edu

ABSTRACT

The position of this paper is that it is worthwhile to invest *now* in use-inspired fundamental research and development leading to a new class of software development environments and methods, in which *software investment opportunities* (in addition to software *capabilities*) are modeled and analyzed explicitly, in support of a *dynamic investment management approach* to software development decision making. Theoretical work in this area has progressed far enough, and at the same time the cost of converting theory to practice has been radically reduced by advances in software development environment technology. The potential payoff on such an investment is a significantly improved capability for both engineers and executives to see, value, and exploit flexibility in software products, projects and processes, leading to significant improvements in decision making and thus in software design productivity.

Categories and Subject Descriptors

D.2.2 [Software Engineering]: Design Tools and Techniques—*design economics*; D.2.6 [Software Engineering]: Programming Environments—*opportunity-driven development*

General Terms

Economics. Design

Keywords

Opportunity-driven, value-based, software development environments, dynamic investment management

1. INTRODUCTION

Recent years have seen increasingly acceptance of the idea that software development should be managed to a degree as an *investment* activity [4]. We see this trend in academic work on *value-based software engineering* [2], including work on *options* in valuing design flexibility in phased projects

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.

FoSER 2010, November 7–8, 2010, Santa Fe, New Mexico, USA.
Copyright 2010 ACM 978-1-4503-0427-6/10/11 ...\$10.00.

and modular architectures [5, 1], in appeals to *mechanism design* as a framework for aligning incentives of individual and organizations with technical needs [3], and increasingly among practitioners, e.g., in the emerging notion of *technical debt* as a metaphor for the carrying costs of design expedients (see related paper in this proceedings).

Research to operationalize this perspective is timely and likely to produce significant payoffs: advancing industrial design and productivity and elucidating fundamentals of design processes. The economic perspective has reached a meaningful level of maturity, visibility and acceptance. At the same time, software development environment technology has evolved to readily support new abstractions. This paper outlines a set of economic abstractions to be supported and argue that it is time to make them a more central focus. The key concept is that of the software *investment opportunity*: a possible task or sub-project whose execution is deemed to have the potential, should the right conditions arise, to increase the value of the project net of the cost of task. The problem this paper addresses is that developers and executives lack adequate support to document, analyze, track, and exploit software investment opportunities.

There are many issues to address. We need a principled, easy to understand economic framework in terms of which everyone from junior developers to senior executives can communicate and reason about software capabilities and opportunities and their value. Distinguishing between *capabilities* (e.g., features) and *opportunities* and making the latter explicit is a key step. Formulating a principled, validated, dynamic approach to deciding when, if ever, to invest is especially important. Work in this area is timely, has intellectual merit in linking design to economics, and has strong potential to produce strong positive impacts on practice.

2. APPROACH

Software engineers and managers connect at the point where decisions are made about what to pay for. A rational decision to make an investment rests on several hypotheses: that there is a *fitness gap*; this gap has created a *potentially profitable investment opportunity*; and conditions dictate that it is now optimal to make the investment. A *fitness gap* is a discrepancy between the state of a product or project and a hypothesized state that is economically better in the assumed *environment*. A fitness gap creates a *potentially profitable investment opportunity* if one judges that there are current or possible future conditions under which it would be profitable to invest in closing the gap. The dynamic decision making problem is to decide when, if

ever, to *exercise* such an opportunity. The goal is to do so in a way that maximally increases project value net of investment costs. Exercising an opportunity yields a new *task* to be carried out by the project team. The question that this paper asks, is what capabilities must software development environments provide to support development processes in which such reasoning about opportunities is central?

Any computation of the value of an opportunity or the payoff on exercising it depends on assumptions about the surrounding environment and how it will evolve over time (e.g., the market's willingness to pay for a particular feature set). Stochastic models of assumed environments are thus required. One must model actual and hypothesized product and project states, fitness gaps, and induced investment opportunities. We also need methods for analyzing and quantifying the present value of investment opportunities, and decision rules for deciding when, if ever, to invest.

The state of an environment changes continually. These changes produce corresponding changes in the values of investment opportunities and must trigger reconsideration of investment decision rules. The implication is that we need to use a dynamic, conditions-driven approach to investment decision making. By making investment opportunities explicit and quantified, this approach will make software investments subject to systematic dynamic management. Such an environment will support both engineering decision making as well as engineer-executive communications about economically critical technical issues many of which (e.g., how architectures create investment opportunities) heretofore have been all but invisible to executives.

3. TERMINOLOGY

I use the term *project* to refer to a combination of software artifacts, process and organization. I use *environment* to refer to the context of use. I do not develop a detailed approach to modeling projects or environments here. For purposes of discussion, I assume an approach in which they are modeled as networks of state variables (or “concerns”, e.g., architectural and API decisions level of demand for a product, etc.), connected by relations representing dependencies among such concerns. Technical characteristics would be modeled as configurations of such variables. A fitness gap would be modeled as the difference between the actual state and a hypothetical better state, e.g., a state with a bug versus one in which it is fixed.

Such models need not be elaborate. Even a simple two-variable model might suffice in some cases: e.g., to compare an actual system having *good features* but an *inadequate architecture* (two concerns) to a better system *good* in both dimensions. The relevant characteristic is modeled by the *architecture* variable: e.g., (*architecture = inadequate*). The gap is that the value is not ideal. The question is whether it adds value to the project to pay to close this gap (to make *architecture = adequate*). Answering this question requires a business analysis. This analysis could be anything from a subjective stipulation to an elaborate, extended net present value or options analysis. The point is, we need development environments that support such modeling and analysis.

4. FITNESS GAPS

Examples of fitness gaps include missing features, defects, inadequate dependability cases, poor architecture. This no-

tion of fitness gaps is broader than the emerging notion of *technical debt*, which focuses on gaps that are hard for non-technical decision-makers to see. I believe, at this time, that an opportunity-based perspective is more technically sound, and largely subsumes the notion of technical debt. The question as whether the debt metaphor is *psychologically* more effective with the intended audience is interesting and remains unanswered.

To determine whether a fitness gap creates a potentially profitable investment opportunity, one must see the project in its environment. A project that is *good enough* even if technically imperfect might be perfectly fit for its assumed environment, in that no technical improvement would improve its value. Throw-away prototypes are in this category. A technically perfect prototype could cost more yet have no more value. On the other hand, a defect in a mission-critical system would create an investment opportunity, in the sense that a hypothetical, corrected, system would have greater value.

The interesting cases are the ones that are not so clear. For example, developers of a project want to spend several months refactoring to ease future development but executives want to keep delivering features. Such disagreements often fail to converge, and end on suboptimal terms. The approach that is described here would encourage and help each side not just to advocate for the investments it favors, but to explicate how they would increase project value.

5. INVESTMENT OPPORTUNITIES

The mere existence of a gap does not create a *prima facie* case for an investment. In this sense, the technical debt metaphor can be misleading, in that it seems to suggest that there is always (eventually) value to be had in repaying a debt (closing a gap). Closing a gap is better understood as an investment option than as a debt or even as a debt with an option to default. A working but unmaintainable legacy system has fitness gaps but still might present no potentially profitable investment opportunities, because any significant change is too costly or risky. Some defects, which clearly create fitness gaps, are still not worth fixing because the risks would outweigh the benefits. Similarly, a technical debt in the form of poor architecture might not be worth fixing if the future holds little demand for change.

It is very important, however, to understand that a gap that cannot profitably be closed immediately can still create a valuable investment opportunity—if the environment might change in a way that would make it profitable to close it. An example is a change in the environment for what started as a throw-away prototype. As soon as a decision is made to develop the prototype into a product, the imperfection that were not worth fixing in the prototype become opportunities that are immediately profitable to exercise.

The current generation of task-centered environments does not support reasoning in terms of environments, gaps, investment opportunities, and dynamic investment decision making. This paper presents a hypothesis that it is *now* worth closing *this* gap. In particular, doing so would enable technical people to make better decisions; it would make the value of the flexibility created by a set of investment opportunities clear both within and outside the technical team; it would also improve managerial decision making in matters of technical importance; and ultimately it would increase the value of investments in software.

6. POSITION

What developers need now is a simple, holistic conceptual framework—a narrative—supported by modern software development environments, within which to reason about the economic issues that must be considered to make rational decisions. The position of this paper is that *evolving fitness gaps between systems and their environments create valuable investment opportunities that should be exercised optimally through a dynamic investment decision making process*. This position leads to a practical framework within which such concepts as present value, real options and technical debt can be organized for practical use. It promises to lead to a new kind of software development environment and process that more effectively supports systematic documentation and management of investment opportunities.

The narrative also appears to have the potential to catalyze deeper shifts in the way we think about software projects. It could, for example, change how we think about *requirements*. The term *requirement* connotes a joint assertion that (a) there is a gap to be closed, and (b) it will be profitable to close it! In reality, even if one is right about (a), the validity of (b) is a separate matter altogether. Rather than *requirements*, a rational project manager seeks to discover or construct *profitable investment opportunities*. Software development might therefore be re-conceived not in terms of requirements, but rather in terms of a search for fitness gaps that are worth closing.

This paper asserts that there is now a significant *fitness gap* between current generation software development tools and methods and a hypothesized class of better tools and methods—ones that support opportunity-driven development. Effective decision making requires systematic consideration of economics, yet the state of the art in tools and methods mainly puts tasks and artifacts at the focus of concern. Elevating tasks to first-class status in software development was a real advance; but it does not go far enough. All the economic reasoning that leads to a selection of tasks is left unsupported. Tasks are not opportunities but only reflect that set of opportunities already being exercised. A task backlog partially represents opportunities but does not come with any economic analysis or decision support. Today's environments provide engineers and executives with a highly incomplete picture of the situation they need to understand.

This fitness gap create a profitable opportunity for an investment in research and development, both because the payoff seems likely to be high, and because the cost is now quite manageable. Past work in two areas leaves us in a strong position to close this gap. First, a decade of work on value-based software engineering has laid the intellectual groundwork. Second, advances in extensible software development environments provides a quick path to operationalizing these ideas. To be concrete, we have an opportunity at relatively low cost to design, build, evaluate, and quickly refine software development tools, environments, and processes by building on such platforms as Eclipse and IBM's Jazz. What Mylyn and Jazz did for the notion of task-centered development, we now need to do for investment-opportunity-centered development. The combination of a simple notion of investment opportunity, analogous to task, with analogous support for explicit representation and tracking within modern development environments, provides a distinct possibility for a high potential impact at a modest cost and with manageable technical risks.

7. EXAMPLE

To make it a little clearer what one would see in such an environment, consider a simple example. A company hypothesizes that the market is reasonably likely (e.g., 50/50) to pay enough for a new kind of software product to generate a significant profit. There is no product today, and so there is a fitness gap between what exists (nothing!) and what is hypothesized (as might be described in a set of "requirements.") This gap creates investment opportunities, among which are two possible projects. One project (P1) would implement the required features and also invest in a clean, extensible architecture to enable long-term sustainability of the system. The cost of this project at time $t = 0$ is estimated at \$2 million. The market uncertainty would then be resolved at time $t = 1$, when the market decides whether or not it likes the product. If the market likes it, the company is projected to receive revenues with a value of \$4 million, for a profit of \$2 million. If the market doesn't like it, the company will receive revenues worth only \$1 million, for a loss of \$1 million. Each of these outcomes is deemed equally likely, and so the net present value of this project at time $t = 0$ is $0.5 * -\$1,000,000 + 0.5 * \$2,000,000 = \$500,000$. There is clearly a profitable investment opportunity here.

However, it's not the only opportunity. Being clever, the managers consider a second project strategy (P2). The idea is to invest enough in a first stage of a project to implement the capabilities that the market needs to see to decide whether it likes the idea. The strategy is to build a quick-and-dirty product, without investing the resources needed for a clean and extensible architecture. Suppose this first stage costs only \$1 million, due to savings on the architecture sub-project. If, on seeing the initial version of the product, the market decides it likes it, then the plan is to invest in a re-engineering project to impose an architecture that will be good for the long term. Because this was not done initially, the cost is higher: this project will consume \$1.5 million. On the other hand, if the market doesn't like the product, the company will simply decline to invest in an improved architecture. In this case, there is a 0.5 chance of a profit of $-\$2.5 \text{ million} + \$4 \text{ million} = \$1.5 \text{ million}$ and a 0.5 chance of a wash $-\$1 \text{ million} + \1 million . The net present value of this project at time $t = 0$ is \$750,000, which is significantly higher than for P1.

In other words, in the environment at time $t = 0$, a project to produce an architecturally inadequate system has a higher present value than a project to build a technically excellent system. What's happening is that the opportunity to invest in an architecture later, at a higher cost, is worth more than having the architecture immediately. If these are the only options, then we have a situation in which the technically imperfect system is the most fit (most valuable). It's the right system to build at $t = 0$, and the right way to model it is as a system with given capabilities *and* an opportunity to make a follow-on investment in systems re-architecting.

At time $t = 1$ the environment has changed. In the unfavorable future environment (low demand), there is no profitable investment opportunity. On the other hand, in the favorable future, the architectural characteristics of the quick-and-dirty solution are no longer well matched to the environment (many future change requests), and so it makes sense to exercise the latent opportunity to re-architect the system (to reduce significant future change costs). Thus we see a dynamic investment decision-making process in action.

While shortchanging the architecture in the first phase could be considered as incurring a technical debt, there is in fact no business case at all for “repaying the debt” until the uncertainty about the future environment is resolved. Moreover, no point does one does have an *obligation* to repay (which is the connotation of the word *debt*, but rather an opportunity to invest in a forward-looking refactoring project. Rather, developers should think in terms of opportunities: rights *without obligations* to make future investments on terms that are understood today. Opportunities are thus technically *options*. We use the term *opportunity* instead, because the *option* term suggests that arbitrage-based real option pricing techniques might apply. In an engineering design setting, we believe they often do not apply, because basic assumptions behind these techniques do not hold (e.g., the existence of replicating portfolios or that uncertainties are characterized by certain kinds of stochastic processes).

What would the developers and managers see in their software development environments in this example? First, the $t = 0$ environment would be modeled, including the estimated probabilities of future events (the market likes, or does not like, our product). This could be done with a simple event tree. Second, key technical characteristics and fitness gaps would be modeled: including lack of an implementation of the required features, and lack of an architecture for the long haul. Third, the key investment opportunities created by these gaps would be identified and models as entries in the environment data base: P1 and P2 and an indication that they are mutually exclusive. Fourth, the business cases the I just laid out would be developed for each of these opportunities. Finally, the one (if any) that most increases project value (P2) would be selected, and a task to develop it would be entered into the task database. At time $t = 1$ a critical event occurs: the market decides, resolving the value of the random variable that models demand. At this point, the environment model is updated. This change in environment would lead to a re-evaluation of fitness gaps and the values of the investment opportunities. In only one of the $t = 1$ futures (high demand) would there be a business case for investing in architectural enhancement. In that future, the refactoring opportunity would be exercised and a corresponding task would placed in the development system.

8. CHALLENGES AND RISKS

The vision presented here would create numerous challenges. First, tracking changes and updating assumptions and the state of the environment is not free. On the other hand, this generally has to be and is done, albeit implicitly and imperfectly. Second, the framework requires answers to the question, “What is the present value of an opportunity to invest in, e.g., architectural qualities, especially if it has payoffs only in an uncertain future?” Here, a pluggable framework is needed to accommodate a range of modeling approaches. Third, while opportunities can lead to tasks, not all tasks trace back to opportunities. Some are simply sub-tasks without separate business cases. This proposed framework also does not confront the many questions that arise in any attempt to build a business cases in the face of an uncertain future. What are the probabilities on the event trees, and where do they come from? In this case, In many cases we can do no better than to use well informed sub-

jective probabilities. Here, too, the framework here should accommodate a wide variety of estimation approaches.

Rather than having a task list at the heart of a development environment, tomorrow’s developers and managers will opportunity lists with associated investment cases, derived from consideration of changing and uncertain environment, present and possible future states of nature, and business cases at widely varying levels of detail and sophistication. A critical observation is that opportunities create flexibility to make investments in the future as conditions warrant; we can model the *present value* of the flexibility to make such investments in the future; and we can develop sound decision rules about when, if ever, to invest.

9. CONCLUSION

Software product, project and process decisions are ultimately driven by economic calculations. A problem is that decision makers generally do not have an adequate picture of all important sources of value. The value of a software product derives from the market’s willingness to pay for its *capabilities and* for the opportunities it affords for potentially profitable future investments: to correct a system (e.g., fix a bug), reduce uncertainty (e.g., more testing), change capabilities (e.g., new feature), or change its opportunity set (e.g., through refactoring). Capabilities are visible both to development teams and to executives and the market, and their value is tangible, even if uncertain. Opportunities, are generally not as visible to executives or the market. They arise from design architectures and other technical aspects of a system. Moreover, their value is intangible, because it cannot easily be determined by immediate market tests, but is instead contingent on uncertain future states of nature. But opportunities do have present value. Indeed, the value of the opportunities created by a system can be significantly greater than the value of its current capabilities. A rational approach to software decision making demands that we make the existence and value of opportunity sets visible and tangible in software development, and that we then work to exploit them dynamically in an optimal manner. This paper proposes that one way to do this is in the development environments that developers use, and in the executive-visible dashboards that these systems support.

10. ACKNOWLEDGMENTS

This work was supported in part by grants 0613840 and 1052874 from the National Science Foundation. I thank the participants in the Software Engineering Institute’s 2010 Technical Debt workshop for providing me an opportunity to present and work out these ideas.

11. REFERENCES

- [1] C. Baldwin and K. Clark. *Design rules. Vol. 1: The power of modularity*. MIT Press, 2000.
- [2] S. Biffi, A. Aurum, B. Boehm, H. Erdogmus, and P. Grunbacher, editors. *Value-based software engineering*. Springer, 2005.
- [3] P. Feiler, R. P. Gabriel, J. Goodenough, R. Linger, T. Longstaff, R. Kazman, M. Klein, L. Northrop, D. Schmidt, K. Sullivan, and K. Wallnau. *Ultra-Large-Scale Systems: The Software Challenge of the Future, Study Report*. Software Engineering Institute, 2006.

- [4] K. Sullivan, P. Chalasani, S. Jha, and V. Sazawal. Software design as an investment activity: A real options perspective. In c. e. L. Trigeorgis, editor, *Real Options and Business Strategy: Applications to Decision Making*. Risk Books, 1999.
- [5] K. J. Sullivan, W. G. Griswold, Y. Cai, and B. Hallen. The structure and value of modularity in software design. In *ESEC/FSE-9: Proceedings of the 8th European software engineering conference held jointly with 9th ACM SIGSOFT international symposium on Foundations of software engineering*, pages 99–108, New York, NY, USA, 2001. ACM.