

Towards Estimating Software Value

by

Howard Baetjer, Jr., Ph.D.
Lecturer, Towson University

hbaetjer@towson.edu
410-830-4658

A paper presented at
the Los Angeles Software Improvement Network (LA SPIN)
October 25, 2000

Foreword

I am an economist by training, although I have graduate degrees in English literature and political science as well as economics. My great love is teaching economics -- helping people come to understand economic processes and principles. Within economics, I am fascinated above all by the general question of *how* human beings have made themselves so productive, and how we may make ourselves more productive still.

Think of the progress humanity has made in its standard of living in the last couple of lifetimes -- at least in the predominantly market economies -- with pharmaceuticals, aeronautics, electronics, computation and so on. We take for granted traveling between Baltimore and Los Angeles in a few hours. We sit comfortably, 35,000 feet above North America, eating a hot meal, listening to Mozart in stereo. We are unconcerned that a few feet away the wind is screaming by at 550 knots, 25 degrees below zero. Our pilots know where they are to within a few *meters*, thanks to global positioning systems. The navigational computers give them the time to touchdown at LAX to the minute (barring traffic delays!). When we stop to think about it, it is almost fantastic. *How is it that we have become so productive? What are the keys to human productivity?*

In pursuing answers to these questions, I have focused on software development for three reasons. First, software appeals to me. With each generation of each word processor I have used since EasyWriter on the original IBM PC, I have marveled at the creativity embodied in those tools and been grateful for the capabilities they put in my hands. I gratefully reconcile my checkbook in five minutes with Quicken, remembering the frustrated hours I used to spend finding errors by hand. And now I teach over the internet: My students "mark up" great works of economics, then pass their comments on to others for further commentary, using Folio VIEWS hypertext software. They exchange the VIEWS "shadow files" which contain their annotations using a web browser or ftp package. We hold our "class discussions" via conferencing software built in Lotus Domino. And how would I keep in touch with my friends from summer camp without email? Software has transformed my life for the better.

Second, there is no category of capital good in our world that is more important. Increasingly, software is helping with the production of everything.

Third, with software there is no physical matter to obscure the essence of any human creation -- the knowledge embodied in it. With an automobile, the steel and glass and paint confuse us into thinking that the value is in that physical stuff, when in fact the value is in the design, the configuration of that steel and glass and paint. With software, there is no steel, glass, paint, or any matter whatever. Software is pure design. As such, it helps us focus on the essence of value, and, therefore, how it may be created.

So much for a quick description of my interest in your field. I write as an outsider, but as one who has thought a lot about the economic nature of software and software development. I hope you will find what follows useful. It is adapted from a consulting

report I wrote for a large firm some years ago. Please note that a lot of it has been taken more or directly from my book, *Software as Capital* (IEEE Computer Society, 1998).

Contents

FOREWORD	I
CONTENTS	III
INTRODUCTION	V
Purpose	v
A note on presentation	v
Plan of the study	v
PART I. THE NATURE OF ECONOMIC VALUE	1
Subjective value -- "value to whom?"	1
The value of capital goods	2
Value and profitability	7
PART II. THE NATURE OF CAPITAL	12
Embodied knowledge	12
Knowledge is of the essence	14
Varieties of knowledge embodied in capital	15
The division of knowledge in capital goods	17
PART III. SOFTWARE DEVELOPMENT AS A SOCIAL LEARNING PROCESS: IMPLICATIONS FOR VALUE CREATION	ERROR! BOOKMARK NOT DEFINED.
Introduction	Error! Bookmark not defined.
Preliminary note - software development is design, not manufacture	Error! Bookmark not defined.
Discovering what the software must "know": the value of an interactive, iterative development process	Error! B
Understanding the evolving design: the value of a rich development environment	Error! Bookmark not defined.
Communicating about the design: the value of code that is similar to natural language	Error! Bookmark not defi
Coping with constant change: the value of modular design	Error! Bookmark not defined.

Introduction

Purpose

The purposes of this study are:

1. to identify the crucial determinants of software's economic value both to customers and to developers and
2. using this analysis, to recommend approaches for developing metrics that will help software developers estimate how well they are creating value in all aspects of the development process.

This effort aims beyond developing "software metrics" as such, which usually focus on code. Some aspects of the software development process that are crucial to overall value creation cannot be measured. They must be estimated, or judged, on the basis of an understanding of the risks and rewards of those aspects of the business. This study aims to lay the groundwork for developing reliable means of estimating value in various aspects of the development process. This study makes no judgments as to the feasibility of developing value estimators for any of the determinants of software value identified. Instead it focuses on identifying what developers ought to measure if they can.

A note on presentation

From time to time there is information I wish to include as useful clarification or amplification of the main points, but which nonetheless is not central to the presentation. Such information I have put in italics in indented paragraphs. This information may be skipped without any important loss of meaning.

Plan of the study

Parts I and II lay the economic and theoretical foundations of the study, examining respectively the nature of economic value and the nature of capital goods, including software. Part III examines the software development process as a social learning process, and draws implications for where and how value is created in that process. Part IV presents recommendations.

Where discussions in Parts I-III are relevant to particular recommendations in Part IV, boxed references direct the reader to those recommendations.

Part I. The Nature of Economic Value

This part of the study lays out some of the vocabulary and concepts necessary for understanding economic value in general, and the value of capital goods such as software in particular. A basic assumption is that one can best understand the value of software by seeing software in its economic role as a kind of capital good. Accordingly, what follows gives a lot of attention to the nature of capital goods and their role in the evolving ecosystem of production

Subjective value -- "value to whom?"

A foundational principle of modern economics is that value is subjective. That is, nothing has inherent value. Value is "in the eye of the beholder"; it is in the opinion of the individual doing the valuing.

An key consequence of this principle for software developers is that it is pointless to think about or talk about your software's value in and of itself. It has *no* value on its own. It has value only to particular people and organizations -- to you and to your different customers. And these valuations will differ, often dramatically. To some customers, one of your products may be essential to their success and hence extremely valuable. Others may have no use for it and hence not value it at all. To your own company, that product will be more valuable if it is easy to maintain and customize for a large variety of customers. On the other hand, if it has too small a market, or it is too difficult to adapt to new market opportunities, it will be less valuable. Accordingly, it is essential to think in terms of value to some person or organization. Always ask, "value to whom?"

Evaluate your software in two general categories:

1. its value to your customers, both present and potential, and
2. its value to your own company.

As we will discuss in more depth below, your software's value to your own company is entirely dependent on, but not the same as, its value to your customers. The value to your company will depend on how many customers (may possibly) value a product, and how much.

N1, N2, p. 22

A note on how costs affect value: How do your costs of production influence your software's value? Again, ask, "value to whom?" Your costs will dramatically influence your software's value to you. Indeed, if your costs are too high, your software will have negative "value" to you. On the other hand, if you succeed in lowering your costs through your current efforts, your software will have correspondingly greater value to your own company because it will be a source of greater profitability.

Your costs are entirely irrelevant to your software's value to your customers, however. Customers will value your product only for what it does for them, not for what it costs you to put it in their hands.

The value of capital goods

Almost every software package is a kind of *capital good*. Capital goods are “the produced means of production,” the tools, raw materials and intermediate goods used in production processes. In this section we look at some basic economic theory useful for understanding what determines the value of capital goods, including software.

Virtually the only kind of software that is not capital is entertainment software -- games and images that are pure consumer goods, meant to be enjoyed directly. We'll ignore this kind of software, and pay attention only to software capital goods.

The ecosystem of production

Software is not useful, hence not valuable, on its own. Like every other sort of capital good, it must work together with other capital goods and people in producing consumer satisfactions. As we will discuss below, consumer satisfactions are the ultimate source of all economic value. To understand software's value, we have to understand its place in productive processes. The concept of the “ecosystem of production” is important to this understanding.

CR 1, p. 20

What we will refer to as the ecosystem of production economists usually speak of as “the capital structure” or “the structure of production.” We mean by these terms the complex and far-flung pattern of interacting tools, processes, and raw and intermediate goods that people use in producing things. The trouble with these standard terms is that the word *structure* suggests something fixed and unchanging. But the complex of productive relationships in the economy, like a vast ecosystem of overlapping food chains, is constantly evolving. The people within it continually develop new tools, processes, and corresponding raw and intermediate goods to feed them. Old processes and tools become obsolete; industries die; new technologies rise. We speak usefully of market “niches” that are open for a time before the economy evolves further. In this study we will use the term *ecosystem of production* to help us see that the system is constantly in flux, constantly evolving.

Some terminology for thinking about capital

Capital goods v. financial capital

Software is a kind of *capital good*. *Capital goods* need to be distinguished from the other major form of capital, *financial capital*. Financial capital is the money available for investment in productive activities, or the financial instruments (which may be bought and

sold) representing such investment. By contrast, *capital goods* are the actual means of production themselves, in which money has been invested.

Categories of capital goods: fixed capital and working capital

There are two categories of capital goods; the first is *fixed capital* or, sometimes, "producer durables" -- tools. These we use over and over again to help us accomplish the transformations that constitute production. The other category is *working capital*, raw materials or intermediate goods, or goods in process. These are the goods that get incorporated into products and become part of them.

Examples come readily to mind when we envision a production process. In the steel mill the machinery is the fixed capital, the iron ingots and molten metal are the working capital. In a bakery, the baker's oven and rolling pin are the fixed capital, the flour and dough are the working capital. In an office context, the word processor, spreadsheet and database management software are fixed capital, and a company's texts and financial data are working capital. They are processed by the word processors and spreadsheets into, say, financial reports.

Software development has historically used only a little fixed capital and very little working capital. For programmers using early programming languages, for instance, only fixed capital in the form of a programming language and its compiler were available. Other fixed capital for software development -- tools such as browsers, debuggers, diagramming tools, version control systems, screen painters and the like -- were unavailable. More significant, there was almost no working capital, no already-written and tested building blocks to work with: The programmer started with a blank screen.

That picture has changed and continues to change for the better. Today's programmers have their languages and compilers, and also an increasing range of additional tools (fixed capital) to work with. Perhaps more significant, they increasingly have working capital in the form of pre-defined classes, design patterns, templates, frameworks and other components which they can use directly or adapt to their purposes. The economic value of such working capital can scarcely be underestimated.

WC 8, p. 24

When categorizing capital goods along the lines of fixed capital and working capital, be aware of your perspective, because depending on one's perspective, one will categorize the exact same chunk of code differently. Your C++ compiler is fixed capital for you. But for the producer of that compiler, it is the end product.

The general theoretical point is that the same item plays many economic roles, depending on how we view its position in the ecosystem of production, and what part of that ecosystem we are concerned with at the moment.

Human capital

Human capital is the skills, experience and capabilities that accumulate in particular human beings. There are few capital goods that can be productive by themselves, without skilled

CR 1, p. 20

people operating them. Rather, fixed capital, working capital and human capital must be used in combination in production. Software's value depends greatly on how well it fits with the human capital with which it must work.

Production processes

Capital of all kinds is valuable only when put to work in particular production processes for which they were designed (or can be adapted). While this may seem to go without saying (a word processor, for example, obviously won't work to manage financial accounts), it is important to remember that the value of software is largely dependent on the quality of its fit with the production processes it must serve. A simple, limited application that ideally fits into a customer's production processes is often more valuable to that customer than a complex, brilliant, powerful application that does not fit well.

CR 1, p. 20

Orders of goods

A useful way to view of the ecosystem of production is in terms of what economists call "orders" of goods. In this view, consumer goods are called "goods of the first order." The capital goods that serve in producing them are called goods of ever higher orders according to how distant from final consumer goods they happen to be in the production chain. As the ecosystem of production lengthens over time, people develop tools for producing tools for producing tools. Every step up the chain is a step to a higher order. Milk, for example, a consumption good, is a good of the first order. We might treat the cow and the hay she eats as goods of the second order, the field in which the hay is produced as a good of the third order, the irrigation system that waters the field as a good of the fourth order, and so on. There is nothing significant about the particular numbers; what is useful is the concept of higher- and lower-order goods, which form chains of inputs and outputs.

Software developers produces higher-order goods -- software tools that help other companies produce their various goods and services. In producing their software, they use goods of still a higher order -- fixed capital such as compilers and development environments, as well as working capital such as templates, frameworks and other components from which applications can be assembled.

DC 7-9, p. 26

An important principle here is that the better the capital goods a software developer uses in its own stage(s) of production, the better and more cheaply it may produce the software tools it supplies to its customers at the next lower stage. Low-cost development¹ depends on the availability and use of both high-quality fixed capital and appropriate working-capital inputs. The fixed capital is the various tools in the company's different development environments. The working capital is the templates, frameworks and other components with which development teams may assemble new applications quickly and inexpensively.

DC 1, p. 25

¹ See the work of Robert Charette on "lean development" described in, e.g., *Application Development Strategies*, Vol. X, no. 11, November, 1998.

With this observation we get to one of the main determinants of human beings' quality of life: The higher the quality of the capital goods in a society's ecosystem of production, the better the fit among those capital goods, and the more stages of production in which capital inputs can be refined and specialized; the higher the standard of living people will enjoy.

Imputation of value from consumer goods to higher-order goods

The ultimate source of the value of capital is consumers at the ends of production chains. The value consumers place on the consumer goods determines the value of all capital goods that help to produce those consumer goods. The value of capital is accordingly derived value. The value of capital goods is derived or "imputed" from consumption goods, up the chains of production.

Consider, for example, the value of, say, software for a global positioning system (GPS). The ultimate source of its value is the satisfaction that consumers place on final goods and services, like fish for dinner or air travel. Those who want fish for dinner will pay fishermen to bring it to them. Those who want to travel fast will pay airlines to carry them. Fishermen and airlines like to keep down their costs and risks with precise navigation, hence they are willing to by GPS systems that help them do so. The value of GPS systems is in this way derived from people's demand for fish and air travel (and the thousands of other goods and services to which global positioning can contribute). (If human beings, for some unthinkable reason, stopped wanting fish or air travel, the value of fishing boats, airliners, and GPS software would immediately plummet.)

A important business implication of this principle is that the value created by software developers and all capital goods makers is limited by the value created for consumers down the various production chains to which the developer contributes.

Capital complementarity

The vertical chains of production that we have just described are one important aspect of the ecosystem of production. In these vertical chains, a set of inputs at one stage is transformed into an output, which then becomes an input at the next stage, and so on.

Another aspect of the ecosystem of production is the horizontal relationships that hold among different kinds of capital that must be used together in any one stage. We refer to these relationships as capital *complementarities*. They are very important in determining the value of any capital good. For illustration, let us go back to the example of hay, which feeds milch cows and thereby becomes milk. To transform the still-growing hay in the field at one stage of production into bales that can be carried to the cow at another requires an intermediate harvesting stage. Harvesting by recent methods requires at least five complementary capital goods: the hay itself, the cutter, the baler, a tractor to pull the baler,

and fuel for the cutter, baler, and tractor.² For efficient harvesting, the cutter must be appropriate to the particular kind of hay, the baler must be appropriate to the windrow the cutter lays down, the tractor must be powerful enough to pull the heavy baler, and the fuel must be appropriate to the kind(s) of engine(s) in the cutter, baler, and tractor. If we try to use diesel fuel in a gasoline engine we get no production at all; if we use a cutter designed for four-foot high alfalfa on eighteen-inch high grass, we waste a terrific amount of fuel and power; if we try to bale a heavy windrow of alfalfa with a small baler designed for light grass, we'll choke the poor machine, and so on.

The essential point is that any capital good we attempt to use in combination with others must fit well with those others in order to be valuable. A fit of minimum quality is essential. The better the fit, the greater the value.

Using our above example of a GPS package we can make the same point. The software must run on particular, specialized machines and operating systems, and it must fit reasonably well with the different needs of various fishermen, airline pilots, and others. The better it does, the greater its value to those customers.

CR 1, p. 20

(Co)evolution in the ecosystem of production

Perhaps nothing has more importance to the value of capital goods over time than the evolution of the ecosystem of production. Consumer tastes and sophistication, production processes and techniques, and (complementary) capital goods and related technologies all change, in our day almost continuously. This means that every kind of capital good occupies a changing niche in the ecosystem of production. In some cases that niche may enlarge, in some cases it may multiply into many related niches, in some cases it may disappear entirely. But it is unlikely to stay the same for long.

For this reason, in order to maintain (or increase) its value in the ecosystem of production, any kind of capital good must itself evolve in order to maintain its fit with the other capital goods and processes that define its niche. This means that change tolerance is fundamentally valuable.

CC 1, p. 22 DC 4, p.25 DC 12, p. 27

The evolution of the ecosystem of production is not a movement toward some particular endpoint, or even in some particular direction. There is no predicting what industries or technologies will become important, in what order. Evolution is necessarily *coevolution* of the different elements of the system. In the ecosystem of production, this means that which capital goods become more useful and which become obsolete at any time is determined by what *other* capital goods happen to be developed also, and what other technologies happen to be discovered.

² This description of hay harvesting comes from my own experience as a ranch hand for a brief time on a large alfalfa hay operation in Nevada, USA.

This fact suggests a range between leader and follower in the evolutionary process. To the extent that a particular developer can get out ahead of technological developments and drive the evolutionary process by offering new and desirable products, it can be the force for changing others' niches. In this position, the developer would enjoy fairly wide choice of its products and its strategy. At the unattractive other end of the spectrum is the follower, scrambling to adapt its offerings to changes in its niches caused by others. For any company there will be some leading and some following in this coevolutionary process, but to the extent that a developer can perceive and act on new opportunities promptly, it can enjoy the more comfortable, profitable position of market leader.

Value and profitability

In this section we will lay out the relationship between value and profitability. In the process we will derive a general formula for product value. Later we will use this formula to help us categorize sources of value in software and the software development process.

Note that the software any developer produces can and should play two economic roles. In one role, it is a product for sale to customers. In this role it is a source of immediate revenue. Its value in this role is obvious.

In its other role, any developer's software is, or can be, a capital input for other products it develops in the future. In this role, software developed primarily for one particular product today is a source of potential future revenue in other products tomorrow. This role is important. Sometimes software developed initially for one project may have greater value as a source of inputs for future production than as a source of revenue from the original product. Developers should strive to develop an entrepreneurial alertness to these kinds of opportunities in their software development personnel. They should also consider developing systems that support generalizing code written for one project into components ready-to-hand for future projects.

WC 1, p. 23

Efficient, profitable software development requires dropping the single project mindset. Everyone in a software development organization should be committed to building capital value for the future, wherever that may be done cost effectively, even as they work on particular projects in the present.

The relationship between value and profitability

Let us take a moment to clarify our usage of the term *value* in order to avoid potential confusion: People use the word *value* in two ways: We use it to refer to the gross utility or benefit that we derive from something (independent of its price or cost). We also use it to refer to the *net* utility or benefit we derive once we have subtracted price or cost. For example, suppose product X saves a customer \$100,000 by reducing that customer's costs. Then we might say the (gross) value of the product to the customer is \$100,000. But suppose the price of the product is \$60,000. Then the net benefit to the customer is

\$40,000. On that basis, we might say that the product's (net) value to the customer is \$40,000.

These are equally legitimate senses of the word *value*. In this study, we will need to use both of them. I shall try to make clear what sense of the term I am using in each case by distinguishing the gross value of software *itself* from the net value or profitability of *producing* or *buying* the software.³ The (gross) value of some software itself, for sale or use in subsequent production, is a matter of the revenue that may be derived from it once it has been produced. The (net) value, or profitability of *producing* that software, by contrast, is a matter of both revenue and costs of production. Similarly, the (gross) value of a software package to a customer, once that customer has purchased it, is a matter of how well that software helps the customer increase its revenues or decrease its costs. The (net) value to that customer is that benefit less the purchase price.

A formula for software value (in the sense of contribution to profit)

Profit (or loss) is yield minus cost. To judge profit (or loss) accurately, one must calculate the total yield -- all the value derived from taking a course of action -- minus the total cost (including price paid, necessary training, associated expenses, the value of other opportunities foregone, and every other expense or cost) of that course of action. Let us use this insight to derive a formula for the value of software. We will give most of our attention to the formula for the maximum possible value of that software, although we will also note the related formula for the actual value a company realizes out of this larger potential value.

Yield from revenue

The value to a developer of *undertaking* any software development project is that project's contribution to the developer's profit (or reduction of its losses). Hence it is the project's yield less the project's costs. We begin with yield.

As we have said, the value of software is derived from the value that software provides customers. Therefore, to begin figuring the yield of a product, we must immediately consider the value to *customers* of buying and using it. Ultimately, its value to them is a matter of increasing *their* profits (or reducing their losses). Their profit (or loss) is their yield minus their costs. Accordingly, they will value the software, if at all, according to how much they expect it to increase the yield and/or decrease the costs of their operations. Simply put, the value of a software package to a customer is the change in that customer's revenues minus the change in that customer's costs, as a result of using the software. In mathematical notation:

Value of a particular application to a particular customer

$$\Delta_{customer}Revenue - \Delta_{customer}Cost$$

³ In this usage we will be following the valuable habit of the Austrian School of economics in focusing on human action. Value we attribute to goods or services as such, but profit and loss are associated with human action.

For example, suppose, by using the product, a customer is able to improve the quality of service it offers its customers, thereby earning \$100,000 more in revenue annually, and also reduce its costs of operation by \$25,000 annually. $\$100,000 - (-\$25,000) = \$125,000$. Using the product has meant \$125,000 per year to this customer's bottom line, thus the value of the product to the customer is \$125,000.

In theory, that customer would pay any amount less than \$125,000 for that product, because by doing so he could come out ahead by the difference. Suppose, for example, one charged \$120,000 for the product. The net benefit to the customer of its choice to buy and use that software would be \$5,000, the net effect on its profitability.

An implication of this arithmetic is that an enterprise can almost never capture all the value it creates for a customer. It must always sell at a price lower than the total value to the customer in order to induce the customer to buy at all.

It is wise sell at a price that is a healthy distance below the customer's valuation. First, in giving the customer more net benefit, the enterprise is more likely to attract that customer's business in the future. Second, the lower the price the enterprise can offer and still make a profit of its own, the more it discourages competitors from offering a similar product. (And of course it must always match or beat the price of competitors' existing products.) This gets into pricing strategy that is really beyond the scope of this study.

Total potential revenue to a developer from a given software package is of course the sum of the value that software might provide to all possible users of it. In mathematical notation, with N representing the *entire* potential market:

Total potential revenue (= total customer value) from a particular application

$$\sum_{i=1}^N (\Delta_{customer_i} Revenue - \Delta_{customer_i} Cost)$$

Of course the actual revenue a company earns from a particular application is less than this. It is simply the sum of the *prices* paid by each customer. Price must in each case be lower than customer value (in order to induce the customer to buy at all), and, unless the package is tailored for a single customer (or the marketing department has done a remarkable job!), the actual number of customers will be less than the potential number:

Actual revenue from a particular application

$$\sum_{i=1}^n (customer_i Price)$$

where n = actual number of customers

Before leaving the topic of yield as it contributes to profitability, it is worth mentioning that the greatest potential for increasing profitability lies in increasing customer value, rather than in cutting costs. It seems to me that most corporate

efforts to improve profit margins aim at cutting costs. This is perhaps understandable: costs are often very visible, and all sorts of possibilities for reducing them suggest themselves. But there is an obvious limit to the benefit of cost cutting -- once all costs have been cut to zero (were that possible) no more progress could be made.

By contrast, there is no limit to the creation of new value for customers. Human experience shows that new value will constantly be created as long as people are free enough to create. Accordingly, in studying their businesses and deciding what software to build, developers should devote plenty of effort to looking for ways to "surprise and delight the customer." Ideally they will become expert at offering their customers capabilities that those customers have not yet imagined, nor perceived that they need, and generate high profits thereby.

CR 2, p. 20

Yield from addition to capital

In estimating the yield from developing a particular application, we must also consider the value of additions to the developer's capital that result from that development. By additions to capital I mean anything the developer builds up during development of that application that can be used to advantage in future production. This can be human capital, fixed capital (tools) and/or working capital (components, templates, frameworks, design patterns, and so on). These all count as capital so long as the developers believe they will be useful in serving customers in the future.

Among the elements that count (the list is illustrative and by no means complete) are:

- the product design, insofar as subsequent versions of the application may be efficiently and inexpensively evolved from it. (Design evolvability -- change tolerance -- is thus an important aspect of software's capital value.) WC 3, p. 23
- the product design, insofar as it may be efficiently and inexpensively adapted to (a) different, but similar (set of) application(s) or elements of them. (Design adaptability -- another kind of change tolerance -- is thus another important aspect of software's capital value.) WC 4, p. 23
- any elements or components of the product, insofar as they may be useful in future production. WC 2, p. 23
- human capital, skills or experience developed by members of the project team, insofar as that is likely to be useful in future development.

The total possible yield from a particular software development project, then, can be represented as follows:

Total possible yield from a particular software development project

total value created for customers + additions to value of developer's capital

= increase in customers' profit (or reduction in loss) + additions to value of developer's capital

$$= \left(\sum_{i=1}^N (\Delta_{customer_i} Revenue - \Delta_{customer_i} Cost) \right) + \Delta_{DeveloperCapital}$$

where N represents the *entire* potential market.

All valuations are estimates, not measurements

The value of these additions to capital can be only roughly estimated or projected, of course, because the future is uncertain. All capital valuations are necessarily estimates. This is because all capital value derives from the value of future production to which that capital contributes, and no one can know for sure the value of future production.

Cost

The total cost of producing and marketing software applications includes marketing and overhead costs as well as development costs. Marketing and overhead costs are beyond the scope of this study, however; we will concentrate on development costs.

I'll have much to say later on the subject of keeping development costs down. At present, I want to stress one dimension of cost that is far too often overlooked or treated with a helpless fatalism in software development organizations. That is the cost of evolving a design over time as the environment in which it is used changes.

"It is widely estimated that 70% of the cost of software is devoted to maintenance."⁴ Initial costs of development, the costs of getting the first version of an application to the customer, have to be kept in perspective. If what Bertrand Meyer says in the quotation is still even half true, keeping down maintenance costs is of tremendous importance to the overall profitability of producing an application. In order to address and minimize those costs, developers have to accept that change is inevitable. Better yet, they will embrace the inevitability of change as a profit opportunity, and consciously build their applications to be *evolvable*, so that they can maintain (or improve) their fit in the evolving ecosystem of production. Only in that way can they maintain (or improve) their value.

WC 3, p. 23

Leaving further discussion of how to hold down development costs until later, our working formula for software value, then, is as follows:

Total potential profit (or loss) from undertaking a particular software development project

$$\left(\sum_{i=1}^N (\Delta_{customer_i} Revenue - \Delta_{customer_i} Cost) \right) + \Delta_{workingCapital} - developmentCost$$

where N represents the *entire* potential market.

⁴ Bertrand Meyer, *Object-Oriented Software Construction*, Hemel Hempstead: Prentice Hall, 1988.

Part II. The Nature of Capital

In Part III of this study we will look at the process of software development. This Part II sets up that examination by considering what capital, hence software, *is*. The discussion in this part may seem to go far afield from our central concern of what to measure in the software development process. But it is important to take this high-level view of the software you are building, in order to appreciate the subtleties, challenges and opportunities of building it. The discussion is based on insights from the capital theory of the Austrian school of economics.

Unlike conventional, mainstream economics, Austrian economics stresses the role of knowledge in the economy, the importance of time and uncertainty, and the challenge of maintaining coordination among people with very different knowledge and purposes.

Knowledge and capital -- all capital, whether software or hard tools -- are fundamentally related. Indeed, capital *is* embodied knowledge of productive processes and how they may be carried out. This relationship will be very important to our understanding of the software development process.

Embodied knowledge

Carl Menger, the first of the great Austrian economists, stresses the role of knowledge in human economic advancement. Fundamental to his thinking is that *knowledge is embodied in capital goods*. It is not enough just to understand physical laws and processes; we must embody this knowledge in tools and devices with which we can direct those processes to our purposes. He writes, "The quantities of consumption goods at human disposal are limited only by the extent of human knowledge of the causal connections between things, and by the extent of human control over these things" (1981, p. 74). In order to provide ourselves with an ample supply of warm clothing, for example, early humans had to develop the knowledge that wool could be spun into yarn and the yarn woven into cloth. But further, if they were actually to *have* woolen clothing they had to apply this knowledge so as to "control" the wool: to spin it and weave it successfully into cloth. This knowledge of spinning and weaving they built into spinning machines and looms -- capital goods for wool production.

In virtually all human production (other than gathering wild berries in open fields, and even there we often bring a pail or a box to carry them in), we employ capital goods -- tools and intermediate goods -- for the purpose. Much of our knowledge of how to produce is to be found in practice not in our heads, but in those capital goods that we employ. Capital is embodied knowledge.

In particular, capital equipment -- tools of all kinds, including software -- embodies knowledge of how to accomplish some purpose.⁵ Much of our knowledge of how to accomplish these purposes is not articulate but tacit. That is, we can do it, but we can't say in detail how we do it. In the beginning of his classic *Wealth of Nations*, Adam Smith speaks of the "skill, dexterity, and judgment" (p. 7) of workers; these attributes are a kind of knowledge, a kinesthetic "knowledge" located in the hands rather than in the head. The improvements these skilled workers make in their tools are embodiments of that "knowledge." The very design of the tool passes on to a less skilled or dexterous worker the ability to accomplish the same results. Consider how the safety razor enables clumsy academics and engineers to shave with the blade always at the correct angle, rarely nicking ourselves. How well would we manage with straight razors? The skilled barber's dexterity has been passed on to us, as it were, embodied in the design of the safety razor.

Adam Smith gives a clear example of the embodiment of knowledge in capital equipment in his account of the development of early steam engines, on which:

a boy was constantly employed to open and shut alternately the communication between the boiler and the cylinder, according as the piston either ascended or descended. One of those boys, who loved to play with his companions, observed that, by tying a string from the handle of the valve which opened this communication to another part of the machine, the valve would open and shut without his assistance, and leave him at liberty to divert himself with his playfellows. (p. 14)

The tying on of the string, and the addition of the metal rod which was built on to subsequent steam engines to accomplish the same purpose, is an archetypal case of the embodiment of knowledge in a tool. The boy's observation and insight were built into the machine for use indefinitely into the future.

⁵ Hayek writes,

Take the concept of a 'tool' or 'instrument,' or of any particular tool such as a hammer or a barometer. It is easily seen that these concepts cannot be interpreted to refer to 'objective facts,' that is, to things irrespective of what people think about them. Careful logical analysis of these concepts will show that they all express relationships between several (at least three) terms, of which one is the acting or thinking person, the second some desired or imagined effect, and the third a thing in the ordinary sense. If the reader will attempt a definition he will soon find that he cannot give one without using some term such as 'suitable for' or 'intended for' or some other expression referring to the use for which it is designed by somebody. And a definition which is to comprise all instances of the class will not contain any reference to its substance, or shape, or other physical attribute. An ordinary hammer and a steamhammer, or an aneroid barometer and a mercury barometer, have nothing in common except the purpose for which men think they can be used. (1979, p. 44)

Knowledge is of the essence

The point here is more radical than simply that capital goods have knowledge in them. It is rather that capital goods *are knowledge* -- knowledge in the peculiar state of being embodied in some medium, ready-to-hand for use in production. The knowledge aspect of capital goods is the fundamental aspect. Any physical aspect is incidental.

A hammer, for instance, is physical wood (the handle) and metals (the head). But a piece of oak and a chunk of iron do not make a hammer. The hammer is those raw materials plus all the knowledge required to shape the oak into a handle, to transform the iron ore into a steel head, to shape it and fit it, etc. There is a great deal of knowledge embodied in the precise shape of the head and handle, the curvature of the striking surface, the proportion of head weight to handle length, and so on.

Even with a tool as bluntly physical as a hammer, the knowledge component is of overwhelming importance. With precision tools such as microscopes and calibration instruments, the knowledge aspect of the tool becomes more dominant still. We might say, imprecisely but helpfully, that there is a greater proportion of knowledge to physical stuff in a microscope than in a hammer.

Computer software offers the extreme illustration of capital as knowledge. Software is less tied to any physical medium than most tools. Because we may with equal comfort think of a given program as a program whether it is printed out on paper, stored on a diskette or tape, or loaded into the circuits of a computer, we have no difficulty distinguishing the knowledge aspect from the physical aspect with a software tool. Of course, to *function* as a tool the software must be loaded and running in the physical medium of the computer. Nevertheless, it is in the nature of computers and software to let us distinguish clearly the knowledge of how to accomplish a certain function from the physical embodiment of that knowledge.

The distinctness of the knowledge embodied in tools from the physical medium in which it is embodied was brought out humorously, years ago, in a remarkable exchange between two engineers working on a moonshot. One, literally a rocket scientist responsible for calculating propulsion capacity, approached the other, a software engineer. The rocket scientist wanted to know how to calculate the effect of all that software on the mass of the system. The software engineer didn't understand; was he asking about the weight of the computers? No, the computers' weight was already accounted for. Then what was the problem, asked the software engineer. "Well, you guys are using hundreds of thousands of lines of software in this moonshot, right?" "Right," said the software engineer. "Well," asked the rocket scientist, "how much does all that stuff weigh?" The reply: "... Nothing!!"⁶

⁶ This story was told me in a personal conversation with Robert Polutchko of Martin Marietta Corp.

Because the knowledge aspect of software tools is so clearly distinguishable from their physical embodiment, software seems at first very different from other capital goods. But there is no fundamental difference between software tools and conventional tools. What is true of software is true of capital goods in general. What a person actually uses is not software alone, but software loaded into a physical system -- a computer with a monitor, or printer, or plotter, or space shuttle, or whatever. The computer is the multi-purpose, tangible complement to the special-purpose, intangible knowledge that is software. When the word-processor or CAD package is loaded in, the whole system becomes a dedicated writing or drawing tool.

But there is no conceptual difference in this respect between a word-processor and, say, a hammer. The oaken dowel and molten steel are the multi-purpose, tangible complements to the special-purpose, intangible knowledge of what a hammer is. When that knowledge is imprinted on the oak in the shape of a smooth, well-proportioned handle, and on the steel in the shape, weight, and hardness of a hammer-head; and when the two are joined together properly; then the whole system -- raw oak, raw steel, and knowledge -- becomes a dedicated nail-driving tool.

All tools are a combination of knowledge and matter. They are knowledge imprinted on or embodied in matter. Software is to the computer into which it is loaded as the knowledge of traditional tools is to the matter of which those tools are composed. As the circuits of the computer can be reprogrammed to other uses, so the physical stuff of the hammer can be taken apart and reformed for other purposes.

Knowledge is the key aspect of all capital goods, because matter is, and always has been, "there." Mankind did not develop its fabulous stock of capital equipment by acquiring new quantities of iron and wood and copper and silicon. These have always been here. Mankind became wealthy through developing the knowledge of what might be done with these substances, and building that knowledge onto them. The value of our tools is not in their weight of substances, however finely alloyed or refined. It is in the quality and quantity of knowledge imprinted on them. As Carl Menger says in his *Principles*:

Increasing understanding of the causal connections between things and human welfare, and increasing control of the less proximate conditions responsible for human welfare, have led mankind, therefore, from a state of barbarism and the deepest misery to its present stage of civilization and well-being. ... Nothing is more certain than that the degree of economic progress of mankind will still, in future epochs, be commensurate with the degree of progress of human knowledge. (1981, p. 74)

Varieties of knowledge embodied in capital

In the above passage Menger asserts a dependency of economic progress on progress of human knowledge. This sounds simple. In the present context, it suggests a simple

principle for developers to use in increasing the value of their software: Build in more knowledge!

Perhaps it would be simple if knowledge were a simple, homogeneous something which could be pumped into a society, or into a software application, as fuel is pumped into a tank. But knowledge is complex and heterogeneous; it is not all of a kind (Polanyi 1958, Hayek 1945). There are important differences among different kinds of knowledge that have to be taken into account if we are to understand how that knowledge gets embodied in capital goods. Much important knowledge is elusive or hidden, requiring a special effort to capture or uncover.

Articulate v. tacit knowledge

An important distinction among kinds of knowledge is that between articulate and inarticulate, or *tacit* knowledge.⁷ Some of our knowledge we can articulate: we can say precisely what we know, and thereby convey it to others. But much of our knowledge is tacit: we cannot say with any kind of precision what we know or how we know it. Hence we cannot explicitly convey that knowledge to others, at least not in words. The experienced personnel officer cannot tell us how she knows that a certain applicant is unfit for a certain job; she has "a feel for it." The skilled pianist cannot tell us how to play with deep expressiveness, although he clearly knows how. A child cannot learn to hit a baseball from reading about it in a book, although the book might help. A skilled object-oriented software designer has a knack for "finding the objects" that will make an application robust and evolvable, but cannot simply tell others what she does and how.

Latent knowledge

Furthermore, much of what we know *we are not aware of knowing*. In such cases we do not become consciously aware of our knowledge until it is somehow brought to our attention, perhaps by our being asked to behave in a way that conflicts with that knowledge. "Let's code it this way," we are asked. "No, that won't work," we reply. "Why not?" "Well, it won't...," we say, but we can't really say why until we take time to think about it, and become explicitly aware, for the first time, of what we have long known. Such knowledge is both *tacit* and *latent*.

For an illustration of latent knowledge, I remember my high school physics teacher telling our class that we all "knew" the Doppler effect -- that the sound made by a moving object sounds higher pitched to us when the object is approaching, and sounds lower-pitched when the object is moving away. We were doubtful. He smiled and made the sound every child makes when imitating a fast car or airplane going past. Sure enough, the pitch goes from higher to lower -- of course, I knew that; but I had not known that I knew it.

⁷ For a full discussion of tacit knowledge, see Michael Polanyi's *The Tacit Dimension*.

Because the knowledge aspect of software tools is so clearly distinguishable from their physical embodiment, software seems at first very different from other capital goods. But there is no fundamental difference between software tools and conventional tools. What is true of software is true of capital goods in general. What a person actually uses is not software alone, but software loaded into a physical system -- a computer with a monitor, or printer, or plotter, or space shuttle, or whatever. The computer is the multi-purpose, tangible complement to the special-purpose, intangible knowledge that is software. When the word-processor or CAD package is loaded in, the whole system becomes a dedicated writing or drawing tool.

But there is no conceptual difference in this respect between a word-processor and, say, a hammer. The oaken dowel and molten steel are the multi-purpose, tangible complements to the special-purpose, intangible knowledge of what a hammer is. When that knowledge is imprinted on the oak in the shape of a smooth, well-proportioned handle, and on the steel in the shape, weight, and hardness of a hammer-head; and when the two are joined together properly; then the whole system -- raw oak, raw steel, and knowledge -- becomes a dedicated nail-driving tool.

All tools are a combination of knowledge and matter. They are knowledge imprinted on or embodied in matter. Software is to the computer into which it is loaded as the knowledge of traditional tools is to the matter of which those tools are composed. As the circuits of the computer can be reprogrammed to other uses, so the physical stuff of the hammer can be taken apart and reformed for other purposes.

Knowledge is the key aspect of all capital goods, because matter is, and always has been, "there." Mankind did not develop its fabulous stock of capital equipment by acquiring new quantities of iron and wood and copper and silicon. These have always been here. Mankind became wealthy through developing the knowledge of what might be done with these substances, and building that knowledge onto them. The value of our tools is not in their weight of substances, however finely alloyed or refined. It is in the quality and quantity of knowledge imprinted on them. As Carl Menger says in his *Principles*:

Increasing understanding of the causal connections between things and human welfare, and increasing control of the less proximate conditions responsible for human welfare, have led mankind, therefore, from a state of barbarism and the deepest misery to its present stage of civilization and well-being. ... Nothing is more certain than that the degree of economic progress of mankind will still, in future epochs, be commensurate with the degree of progress of human knowledge. (1981, p. 74)

Varieties of knowledge embodied in capital

In the above passage Menger asserts a dependency of economic progress on progress of human knowledge. This sounds simple. In the present context, it suggests a simple

principle for developers to use in increasing the value of their software: Build in more knowledge!

Perhaps it would be simple if knowledge were a simple, homogeneous something which could be pumped into a society, or into a software application, as fuel is pumped into a tank. But knowledge is complex and heterogeneous; it is not all of a kind (Polanyi 1958, Hayek 1945). There are important differences among different kinds of knowledge that have to be taken into account if we are to understand how that knowledge gets embodied in capital goods. Much important knowledge is elusive or hidden, requiring a special effort to capture or uncover.

Articulate v. tacit knowledge

An important distinction among kinds of knowledge is that between articulate and inarticulate, or *tacit* knowledge.⁷ Some of our knowledge we can articulate: we can say precisely what we know, and thereby convey it to others. But much of our knowledge is tacit: we cannot say with any kind of precision what we know or how we know it. Hence we cannot explicitly convey that knowledge to others, at least not in words. The experienced personnel officer cannot tell us how she knows that a certain applicant is unfit for a certain job; she has "a feel for it." The skilled pianist cannot tell us how to play with deep expressiveness, although he clearly knows how. A child cannot learn to hit a baseball from reading about it in a book, although the book might help. A skilled object-oriented software designer has a knack for "finding the objects" that will make an application robust and evolvable, but cannot simply tell others what she does and how.

Latent knowledge

Furthermore, much of what we know *we are not aware of knowing*. In such cases we do not become consciously aware of our knowledge until it is somehow brought to our attention, perhaps by our being asked to behave in a way that conflicts with that knowledge. "Let's code it this way," we are asked. "No, that won't work," we reply. "Why not?" "Well, it won't...," we say, but we can't really say why until we take time to think about it, and become explicitly aware, for the first time, of what we have long known. Such knowledge is both *tacit* and *latent*.

For an illustration of latent knowledge, I remember my high school physics teacher telling our class that we all "knew" the Doppler effect -- that the sound made by a moving object sounds higher pitched to us when the object is approaching, and sounds lower-pitched when the object is moving away. We were doubtful. He smiled and made the sound every child makes when imitating a fast car or airplane going past. Sure enough, the pitch goes from higher to lower -- of course, I knew that; but I had not known that I knew it.

⁷ For a full discussion of tacit knowledge, see Michael Polanyi's *The Tacit Dimension*.

A great deal of the knowledge that is essential to the success of a software application is initially of this kind. Certain basic ways of doing business, for example, are *second nature* to people in a particular company, so much so that they are not consciously aware of them and thus cannot describe them, at, for example, the "requirements stage" of a software development project. As we shall see, this lack of conscious awareness of much of what people know has important implications for how software developers should try to incorporate this kind of knowledge into new software.

CR 3, p. 22

Dispersed knowledge

Another problem with capturing the knowledge we need to build into a software application or other capital good is that human knowledge is widely dispersed. Humans are social creatures who specialize and divide their labor and knowledge. We form groups and teams, as well as arm's-length contractual relationships, in which we depend on one another for different special knowledge. Usually, to create a good software application or machine, tool builders must bring together the knowledge of many.

Incomplete knowledge

A final problem of knowledge in software is that much of the knowledge that is needed in a major project has yet to be developed at all. New insights, understandings, techniques and abstractions must usually be developed in the course of a new application's development. This is where much of the satisfaction and challenge of software development lies: addressing new kinds of problems in new ways. Rarely, on a major project, does the design team already know how to create all the functionality needed. Rarely does the customer know what is possible. A healthy collaboration between designers and clients often creates much new knowledge of what is possible and how it might be accomplished.

CR 5, p. 21

More generally, human beings are always learning more, advancing our understanding of scientific principles and our technology. In an important sense we are always woefully ignorant of what we might accomplish, of what technologies are around the corner, or down the road a number of years. A creative software development organization will constantly be learning how to build software better and better.

The division of knowledge in capital goods

Good tools have productive power because their builders have succeeded in bringing together in useful form a large body of related knowledge, much of which was originally tacit or latent, and which was originally dispersed among many people. Because the knowledge embodied in our capital has come from so many different people with different specialized knowledge, there is a distinctly *social* nature to capital goods and to the

ecosystem of production they support. Most individual capital goods are manifestations of a far-flung *division of knowledge*, an extensive sharing of knowledge and talent across time and space.

Consider modern computer programming. In the earliest days of machines such as ENIAC, the “programmers” were people who physically set switches and connected cables, according to the directions of “the program,” which had been written on paper by someone else. Now we have compilers to direct the setting of the machines’ transistors for us. The compilers have been written by specialists in the technical workings of the machines. In our day programmers work at a much higher level of abstraction than did programmers of old. They need not know how to assign particular values to particular registers in their particular kind of machine, for instance, because the people who built the compiler are virtually present. So are those who built the user interface, the browsing tools, the classes in the class hierarchy, and so on. They are all “there” in the sense that their special knowledge is there, built into the programming environment.

Similarly, consider again our imagined global positioning system software, customized for use by a company that intends to market hand-held GPS systems to boaters. At its best, that software embodies the special knowledge of experts in the satellite navigation domain, and more specifically the maritime navigation domain. It embodies knowledge of the special needs of boaters, such as being able to record a position with a single button-push, in the case of a man overboard. It embodies knowledge of specialists in information presentation (embodied in the GUI). It embodies CR 6, p. 21 knowledge of the workings of the battery-driven computer and small LCD screen on which the software must run. We could go on at length.

The point is that software of any complexity, like all machinery of any complexity, represents a quite astonishing accumulation of diverse knowledge. The marvel is that it embodies it in a form that makes it highly useful to people who personally know almost none of that knowledge.

As Thomas Sowell has observed, “[T]he intellectual advantage of civilization ... is not necessarily that each civilized man has more knowledge [than primitive savages], but that he *requires far less*” (1980, p. 7, emphasis in original). Through the embodiment of knowledge into an extending ecosystem of production, each of us is able to take advantage of the specialized knowledge of untold others who have contributed to it.

Part IV. Implications for what to try to estimate

Following are my recommendations for what developers should try to measure -- or rather to estimate, judge, or assess -- in the software and software development process.

The presentation is keyed to the following formula derived in Part I:

Total potential profit (or loss) from undertaking a software development project

$$\left(\sum_{i=1}^N (\Delta \text{customer}_i \text{Revenue} - \Delta \text{customer}_i \text{Cost}) \right) + \Delta \text{workingCapital} - \text{developmentCost}$$

where N represents the *entire* potential market.

Each particular recommendation is presented under the heading of one of the elements of this formula.

CR	refers to	<u>customer revenue</u>
CC	refers to	<u>customer cost</u>
N	refers to	<u>number of customers in the market for that kind of software</u>
WC	refers to	<u>working capital</u>
DC	refers to	<u>development cost</u>

Where appropriate, the recommendations are also categorized as applicable to

- applications for delivery to customers (whether outside or within the developer's company)
- software capital developed for the developer's company
- the software development process

Reminder: Software development is always designing, in some sense. As a design process, it is fundamentally different from the manufacturing of an established design. Design is by its nature open-ended -- if we knew in detail what the design would turn out to be, the design would be complete already. Because the results of the design process are unknown, the kind of precise measurement and statistical process control that is so useful in manufacturing is largely irrelevant in software development. One can measure precisely and tune the error tolerances of sheet metal being made into a car door because one is entirely clear about what one is making -- the design is established. But with software, once the design is established, all that remains is copying it onto a storage medium or distributing it over a network.

Making software is by its nature always a matter of design, and design is a creative, not a repetitive or predetermined process. This means that the important measurements are mostly going to be imprecise measurements. It may be better to call them judgments or estimates.

Change in customer revenue

Measurements of applications for delivery

CR 1 - Judge the quality of the application's fit in the business niche where it will be used.

The elements of this niche (in the larger ecosystem of production) are the skills and habits of the users and the other tools, software, and equipment with which software must be used. The better the fit, the more valuable the software. Some considerations are the software's fit with:

- business rules
- accounting practices
- operator skills

Remember that the ecosystem of production is constantly evolving. Ideally, a developer's products will anticipate slightly, and even drive that evolution.

CR 2 - Judge how well the product "surprises and delights" the customer.

(See CR 4 for additional discussion) The more developers can go beyond customers' expectations in functionality, ease of use, simplification of processes, and other dimensions of quality, the better.

Note that much of this assessment is likely to occur during customer trials of prototypes and developing designs. As the customer reacts to what the software both can and cannot do for him at the time, designers can learn a lot about what might satisfy them better.

Measurements of the development process

CR 3 - Evaluate how well your procedures draw from users their *tacit* and *latent* knowledge.

There will always be a danger of falling back into the trap of assuming users can articulate what they want. Your development procedures should explicitly assume they cannot. Use techniques such as role-playing, prototyping, and frequent customer interaction with the evolving design to draw from them the knowledge they cannot articulate.

CR 4 - Judge how well you assess users' requirements, both present *and* future.

The economy constantly evolves. Therefore developers should try to anticipate their customers' future needs while it learns their present requirements. To the extent that they can do so, they can "surprise and delight the customer." Often developers will be able to see from their own perspective things that the customer cannot see. How well does the development process assess customer needs, especially in its early stages of figuring out what the customer *thinks* he wants?

The attitude involved here is very different from the stereotypical, stodgy, conservative approach, which leaves it to the customer to state and sign off on his requirements. If developers will develop the skills of looking beyond what the customer thinks he needs to what he is very likely to want, they are likely to win devoted customers.

Note that this kind of requirements assessment may and should also be worked on during any business or feasibility studies done prior to development.

Note also that this kind of assessment is a creative process of a sort that usually benefits greatly from interaction among a team of people with different perspectives.

CR 5 - Assess the smoothness and “bandwidth” of interaction among your customers, your developers, and the evolving design.

How much you learn about what a new product should do depends closely on the interactive “design dialogue.” Set up and evaluate procedures for making this dialogue timely, and for facilitating real-time interactions among designers, customers, and the evolving product.

Remember that because customers often cannot articulate what they want, or what they are dissatisfied with, it is important that your designers actually watch customers using the prototype or intermediate design.

CR 6 - Estimate the number of different people with valuable knowledge to contribute; compare that to the number who actually *do* contribute to the design dialogue.

This will require making judgments as to how many people it is cost-effective to hear from, and then making an effort to see that you do hear from each. One way to hear from many on the customer’s side is to have a developer representative take the evolving product to each one, and watch as he or she interacts with it. The more people you can hear from who might have something valuable to contribute, the better.

CR 7 - Judge the adequacy with which each of these people is able to contribute his or her specialized knowledge to the effort.

The specialized knowledge of various individuals must be captured by developers. Designers must be on hand to listen carefully to the reactions of each, or take time to go over any written comments.

Change in customer cost

Measurements of applications for delivery

CC 1 - Measure change tolerance.

Virtually all applications will need to be changed as customer needs change. How inexpensively can developers make changes for their customers? Alternatively, how inexpensively can the customer make changes for itself?

CC 2 - Assess how easy the application is to learn.

The more easily learned, the better.

CC 3 - Assess how easy the application is to use.

The easier the software is to use, the lower the cost of use to customers.

Number of potential customers

Measurements of applications for delivery

N 1 - Estimate the size of the market for a given application.

Ideally, you do this before developing an application. The greater the market, other things being equal, the better.

The more generally useful the application, of course, the greater the market size. Spreadsheets, for example, have a potential market of nearly every enterprise in the world, because nearly every enterprise can make use of a spreadsheet's functionality.

N 2 - Estimate the size of the market for a given *family* of applications.

This should be done before developing the first in the family, and also before investing in the working capital that will be used in other members of the family. A smaller estimated market means the value of investment in, say, a template for that family of applications will be lower.

Additions to developers' working capital

Measurements of applications for delivery

WC 1 - Assess the degree to which particular applications are written in general terms.

Any application development is an opportunity to capture abstract functionality that can be used in a family of related applications. Ideally, of course, many applications will simply use existing functionality. But where new functionality must be created, developers should measure how well its designers and programmers create that functionality in a generally-usable, rather than narrowly-targeted fashion.

An ideal to shoot for is to develop every new kind of capability to elegantly, and at such a high level of abstraction, so that it will never have to be created again, anywhere, ever.

WC 2 - Assess the general usefulness of particular design elements.

Some design elements will be useful across domains. Such design elements merit refining and importation to other templates and applications in the company repertoire.

WC 3 - Assess the change-tolerance -- the evolvability -- of application designs in terms of how easily subsequent versions may be developed from them.

The more modular and understandable the design is, and the more robust its fundamental abstractions, the more easily the application may be improved.

WC 4 - Assess the change-tolerance -- the evolvability -- of application designs in terms of how easily they may be adapted to different purposes.

The more modular and understandable a design and its various components, the more easily they may be adapted to other purposes.

Measurements of capital accumulated for subsequent software development

WC 5 - Assess the suitability of any functionality that has been or might be purchased from outside.

There is an increasing amount of pre-built functionality available. Using this kind of already-available functionality can save a tremendous amount of pointless redevelopment time, if the software is of high quality.

WC 6 - Assess the quality of components developed in previous projects, and considered for general future use.

It should go without saying that before any functionality is accepted for general use in a variety of projects, this software should be thoroughly tested, and evaluated for the

robustness of its abstractions and the elegance, modularity, and understandability (hence change-tolerance) of its design.

WC 7 - Assess the value of your templates.

Templates that embody all the fundamental entities and relationships in a broad domain have tremendous potential to simplify and streamline software development efforts in those domains. The development and ongoing improvement of these templates should be a major goal.

Templates should be evaluated for their

- comprehensiveness - do they really capture all the essential abstractions of a domain?
- elegance - do they abstract cleanly, so that they can represent any member of the domain?
- understandability

Measurements of the development process

WC 8 - Assess the rate at which your company is accumulating new working capital.

Accumulating working capital is of the utmost importance to a developer's ability to deliver value rapidly and profitably.

Remember that to deserve the name *capital*, any code must actually be useful in future application development (in the judgment of software developers). Any old code tossed carelessly into a database does *not* qualify! The modules must be easily usable; therefore they must be well documented and reliable. They must be of such quality that subsequent developers are eager to use them. To that end, they must capture the abstractions necessary to their purposes. They must be modular.

WC 9 - Assess the extent and quality of investment in existing code.

One important species of this capital accumulation is making *improvements* to existing general templates and other modules, so that they will perform better and better in future projects. Developers should track this process closely.

When new functionality is developed for some particular project it should be evaluated as a candidate for further investment (in quality of abstraction, simplification, and documentation).

WC 10 - Assess the extent and quality of investment in templates.

Templates that embody all the fundamental entities and relationships in a broad domain have tremendous potential to simplify and streamline software development efforts in those domains. Investment in such templates should be continuous; its progress should be carefully tracked.

Development cost

Measurements of applications for delivery

DC 1 - Measure the availability of working capital inputs for the application in question.

This will depend in general on how well the developer has accumulated such capital, of course. How well have you done in building and refining templates and/or other components (such frameworks and even individual objects) for this kind of application? Do you have robust inputs to build with, or must you start from scratch? The availability of templates and components that already provide needed functionality will be a prime determinant of development cost in any new application.

DC 2 - Assess the skill and thoroughness with which the company's designers and programmers use these inputs.

It is pure waste to accumulate templates and other components for various domains and then not use them. Designers and programmers must be trained to use them. Do they use them? How skillfully do they use them? How thoroughly?

DC 3 - Judge the change tolerance of the evolving design during initial development.

By change tolerance here is meant the ease with which substantial change can be made and the cost of such change. Almost all designs need to be changed as designers and users gain experience with the evolving design in the early stages of development. In order for the development process to capture new knowledge quickly, the initial design must accommodate a lot of change quickly. If ever designers say, "We can't do that now; it would break the rest of the design," it is probably time to look for a new development environment that *can* accommodate such change. Getting stuck early is deadly to product quality over time.

DC 4 - Judge the change tolerance of delivered applications.

By change tolerance here is meant the ease with which substantial change can be made and the cost of such change.

Virtually all applications need to be changed after delivery, as customer needs change. How quickly, easily, and inexpensively can developers make changes, including major redesign work? In respect to the software itself, the modularity (object-orientation) and quality of documentation are likely to be two key factors.

(DC 5 - Avoid the optimization trap.)

Be wary of any call to assess how well an application is optimized -- in terms of how well it makes use of machine resources -- for a specific set of requirements. Those requirements will change. There is a steep tradeoff between efficient use of machine resources and

efficient use of human (programming and design) resources. In any but the shortest run, developers will profit by letting their applications run a little more slowly and take up a little more space in order for their basic designs to be more evolvable. A few milliseconds of runtime are a poor trade for man-months of maintenance and redevelopment time.

Of course if every millisecond and every byte of memory are important in some specific application, then the case changes. But only arguments of this kind should be accepted in surrendering change-tolerance for “optimization.”

Measurements of capital accumulated for future development

DC 6 - Estimate the return on investment for each addition to working capital.

This financial measurement should be a long-term, ongoing accounting effort for the purpose of helping the software development organization judge the quality of its investments. How much was invested, for example, in generalizing a particular application into a generally useful template? How much development time and effort was saved, later, through using that template for a variety of related applications? These kinds of calculations should offer guidance as to what kinds of templates and other components it will be profitable to invest in.

Measurements of the development process - tools

DC 7 - Assess how well the tools of the development environment help designers and programmers understand their designs from a variety of perspectives.

Large programs are too complex to be understood as a whole. A variety of tools, offering a variety of views into the design, aid understanding and hence rate of learning.

DC 8 - Assess how rapidly the programming environment gives designers and programmers feedback on what they are doing.

The programmer/designer’s ability to understand her evolving design, immediately and thoroughly, is of tremendous importance in holding down development time and costs. The more lively and timely the “dialogue” between designer/programmer and the evolving design, the more rapidly the designer/programmer will learn what to do. To this end, incremental compiling and detailed debuggers are very valuable. By contrast, programming environments that permit only infrequent “builds,” or that leave programmers in the dark as to where bugs are occurring, are problematical and should be avoided if at all possible.

DC 9 - Assess how well the tools, especially documentation, diagramming, and configuration management tools, help team members coordinate their efforts.

Software development is a social process, a team effort. The quality of the team’s coordination -- the clarity of their different tasks and the timeliness of their information about others’ efforts, is fundamentally important.

Measurements of the development process - code

DC 10 - Assess how well the semantics and syntax of the various design elements help designers think about the evolving design.

“The semantic gap” between different design elements is a source of costly error. Developers should use a development environment that reduces these gaps as much as possible. The more designers can code solutions to problems in terms similar to how they think about those problems, the better.

DC 11 - Assess how well the semantics and syntax of the code help designers and users communicate about the evolving design.

Valuable communication between users and designers will be fostered if the terminology used in the code is similar to the terminology the users’ own terminology for their business systems being modeled.

DC 12 - Judge change tolerance -- the degree to which substantial change can be accommodated and the cost of such change.

All designs need to be changed and designers and users gain experience with the evolving design. Virtually all applications need to be changed after delivery, as customer needs change. How quickly, easily, and inexpensively can developers make changes, including major redesign work? In respect to the development process, the quality of the development environment and the coordination among members of the design team are likely to be two key factors.