

# Towards Estimating Software Value

by

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

[hbaetjer@towson.edu](mailto: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

<b>FOREWORD</b>	<b>I</b>
<b>CONTENTS</b>	<b>III</b>
<b>INTRODUCTION</b>	<b>V</b>
Purpose	v
A note on presentation	v
Plan of the study	v
<b>PART I. THE NATURE OF ECONOMIC VALUE</b>	<b>1</b>
Subjective value -- “value to whom?”	1
The value of capital goods	2
Value and profitability	7
<b>PART II. THE NATURE OF CAPITAL</b>	<b>13</b>
Embodied knowledge	13
Knowledge is of the essence	15
Varieties of knowledge embodied in capital	17
The division of knowledge in capital goods	18
<b>PART III. SOFTWARE DEVELOPMENT AS A SOCIAL LEARNING PROCESS: IMPLICATIONS FOR VALUE CREATION</b>	<b>20</b>
Introduction	20
Preliminary note - software development is design, not manufacture	20
Discovering what the software must “know”: the value of an interactive, iterative development process	21
Understanding the evolving design: the value of a rich development environment	32
Communicating about the design: the value of code that is similar to natural language	37

Towards Estimating Software Value	iv
<b>Coping with constant change: the value of modular design</b>	<b>40</b>
<b>Using the division of knowledge: the value of working capital -- components, frameworks, templates, and design patterns</b>	<b>45</b>
<b>PART IV. IMPLICATIONS FOR WHAT TO TRY TO ESTIMATE</b>	<b>51</b>
<b>Change in customer revenue</b>	<b>52</b>
<b>Change in customer cost</b>	<b>54</b>
<b>Number of potential customers</b>	<b>54</b>
<b>Additions to developers' working capital</b>	<b>55</b>
<b>Development cost</b>	<b>57</b>
<b>APPENDIX CHANGE TOLERANCE THROUGH MODULARITY</b>	<b>60</b>
<b>How modularity promotes evolvability</b>	<b>60</b>
<b>Kinds of modularity</b>	<b>61</b>
<b>Design principles that yield modularity</b>	<b>64</b>
<b>BIBLIOGRAPHY</b>	<b>70</b>

# 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. 54

*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. 52

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. 56

*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 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.

CR 1, p. 52

### 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. 52

### 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. 58

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 development<sup>1</sup> depends on the availability and use of both high-quality fixed capital and

DC 1, p. 57

---

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

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.

*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 buy 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.<sup>2</sup> 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. 52
-------------

### ***(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. 54 DC 4, p.57 DC 12, p. 59
---

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.

---

<sup>2</sup> 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.

CR 4, p. 52

---

## 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. 55

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.<sup>3</sup> 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

---

<sup>3</sup> 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.

$$\Delta customerRevenue - \Delta customerCost$$

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 \$5000, 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

$$\sum_{i=1}^n (\text{customer}_i \text{ Price})$$

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. 52

### **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. 55
- 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. 55
- any elements or components of the product, insofar as they may be useful in future production. WC 2, p. 55

- 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

$$\begin{aligned} & \text{total value created for customers} + \text{additions to value of developer's capital} \\ &= \text{increase in customers' profit (or reduction in loss)} + \text{additions to value of developer's capital} \end{aligned}$$

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

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.”<sup>4</sup> 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

---

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

in the evolving ecosystem of production. Only in that way can they maintain (or improve) their value.

WC 3, p. 55
-------------

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.

## 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.<sup>5</sup> 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.*

---

<sup>5</sup> 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 an 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!!"<sup>6</sup>

---

<sup>6</sup> 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.<sup>7</sup> 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*

---

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

*past. Sure enough, the pitch goes from higher to lower -- of course, I knew that; but I had not known that I knew it.*

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. 54

### ***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. 53

*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 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.

CR 6, p. 53

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 III.

# Software Development as a Social Learning Process: Implications for Value Creation

---

### Introduction

We saw in Part II that software, like all capital goods, is in essence embodied knowledge. Furthermore, much of the essential knowledge that becomes software is initially tacit, latent, dispersed, and/or incomplete. In this Part we look at the implications of these insights for the software development process. There are two fundamental, underlying points. The first is that because major software applications embody knowledge which is initially dispersed among many different people in both the customers' and the developers' organizations, new software development is necessarily a *social* process. That is, it is a process of human interaction through which different specialists work together.

The second main point is that because so much of our useful knowledge exists in tacit or latent form, and because on a project of any significance brand new knowledge must be developed, this learning necessarily occurs in a time-consuming *process*. In this process, different people's knowledge is discovered, articulated, made conscious and explicit, and new knowledge is developed collaboratively.

Accordingly, software development, at least on a project of any magnitude, is inevitably a *social learning process*. This is true even though some misguided software development methodologies call for customers to articulate their needs completely in a requirements document, and for software engineers to produce an optimized design based on those requirements, before a line of code is written. Such methodologies, out of touch with reality, generally produce poor or unusable software at shocking expense.

By contrast, methodologies and approaches to software development that accommodate the inevitably interactive, creative, indeterminate nature of software development are much more likely to produce good software at low cost. The challenge for software development organizations is to work out an approach to development that takes full account of these realities. They must work out an approach to software development *that fosters the social learning process*.

---

### Preliminary note - software development is design, not manufacture

Software development is a matter of producing *designs* for goods, whereas manufacturing is a matter of producing individual *instances* of particular designs. The two differ.

Contrast our common conceptions of producing cars, on the one hand, and of producing software, on the other. When we think of GM "producing cars," we think of their work creating new instances of extant designs. True, GM employs many designers, who *design* new cars, but we don't think of that; we think of the assembly line, spot welding, riveting, bolting, etc.: the hard work of *realizing* these designs -- imprinting the design on metal and rubber and glass so that a new instance of the design -- a new car -- comes to be.

When we think of a developer's work producing software, by contrast, we think of programmers writing code -- creating new designs (or enhancing older designs). True, they may employ people who store the programs onto diskettes, thus in a sense creating instances of the extant designs; but we don't think of that; we think of the late nights at the terminal designing, coding, revising, running, debugging, etc.: the hard work of *creating* new software -- new designs -- specific instances of which will eventually be copied in mass onto diskettes and distributed.

The point here is not that design is unimportant in heavy industries such as automobile manufacturing.<sup>8</sup> Not at all. In fact, design is just as important in such industries as in software. Indeed, by way of example, the design process for the GM-10 line of cars at General Motors was allocated \$7 billion and five years (Womack et. al. (1990), pp. 104-6). The point is simply that the designing of capital goods and what we will call their *instantiation* -- the creation of actual instances of those designs -- are fundamentally different kinds of processes. Instantiation is concerned with the known, design with the unknown. Instantiation is a matter of imprinting a design onto a medium; design is a matter of bringing together and articulating in symbols the knowledge of how to accomplish some productive purpose.

Because design is a process of bringing together and embodying productive *knowledge* in a handy, ready-to-use form, design is a *learning* process. Because that knowledge is of different kinds and widely dispersed among different people and institutions, design is a *social* learning process -- it depends on the interaction of a number of people. Capital is embodied knowledge. *The designing of capital, the developing of the capital structure, is a social learning process whereby the knowledge gets embodied in usable form.*

---

## Discovering what the software must "know": the value of an interactive, iterative development process

### Summary

Because software development is a social learning process, developers should structure its development process to facilitate an ongoing interaction with their customers. This interaction should amount to a dialogue about the evolving design, through which the designers can learn what

---

<sup>8</sup> Indeed, product design in manufacturing industries is receiving a lot of attention. See Wheelwright and Clark (1992), and Womack et. al. (1990).

is wanted and the customers can learn what is possible. Through such a dialogue, the software can come to embody all the knowledge it needs to satisfy the customer.

In the early, high-hype days of computer assisted software engineering (CASE), many hoped that CASE would revolutionize software development by providing tools that would automate the process. CASE enthusiasts misunderstood the process, however. They thought building software was simply a matter of *transforming* knowledge of what a software system was intended to do into the code that would do it. Their unexamined, fatal assumption was that *the knowledge they needed was available* in usable form at the outset. If it were, then in principle they could use their various tools for analysis, design, and code generation to translate that knowledge, in "a step-by-step disciplined process," into the target system.

In fact, the process of software development is *not* one of transforming existing knowledge into code. It is one of eliciting, articulating, discovering, and creating knowledge, as well as transforming that knowledge into code. The knowledge does not and cannot exist in expressible form at the outset. It is dispersed; much of it is tacit; much of it is latent; and it is almost always incomplete. Software development is not so much a process of translating knowledge as of discovering and articulating knowledge. There is the additional problem that much of the knowledge gets discovered, revealed, or created *only in the process* of trying to transform other knowledge into code.

In short, software development can be usefully understood as a process of coming to understand fully what a software system should be. Any software application is born as a rather sketchy, abstract conception; it matures<sup>9</sup> when this vague conception has been satisfactorily articulated in executable code. Thus the process of building a piece of software is a process of coming to understand it fully, in the sense of being able to articulate it in detail. Developing software is a matter of understanding fully what we are trying to achieve.

In this respect, software development is like any other design process. A bridge is fully designed only when the drawings, materials to be used, and other specifications are fully worked out. What the engineers do in designing a bridge is come to understand that bridge fully, so that they can articulate their design in complete detail in the appropriate code -- in their case detailed drawings, materials specifications, etc.

Realistic approaches to software development now use various techniques aiming to *discover* what knowledge is relevant -- what needs and opportunities a new software tool may address, and how. Important among these techniques is *prototyping*. The prototyping process constitutes a kind of dialogue in which all the various people participate whose knowledge must be embodied

---

<sup>9</sup> Or, rather, it reaches a useful stage in its evolution. Because the capital structure around it evolves, this tool must evolve also, or die. In this sense, a particular application is really only *finished* when it is becoming obsolete, and thus not being developed any further.

in the new capital.<sup>10</sup> The medium for the dialogue is the prototype itself -- the emerging design. In a useful sense it is the prototype itself that learns, rather than the human participants, because it is in the prototype alone that all the relevant knowledge may be found in useful form.

Prototyping as such is particularly useful in the early stages of product development, as a means of discovering what the product should be. But the reason prototyping is useful maintains its force even after the prototyping stage: There is usually more that can and should be learned about what the product should be, that can be revealed only through interaction between clients and designers, and also between clients and the developing product.

In order to facilitate the social learning necessary to produce the kind of product the customer wants and needs, developers should put in place an iterative, short-cycle process of incremental development and customer feedback. In a sense, the entire development process should be a kind of extended prototyping process, in that the customer's reactions should be called for regularly, and the design refined accordingly.<sup>11</sup>

CR 3-7, pp. 52, 53
--------------------

### ***Capturing divided, tacit knowledge by prototyping***

Mark Mullin begins his 1990 book, *Rapid Prototyping for Object-Oriented Systems* with this loose definition of rapid prototyping:

This book deals with the concept of *rapid prototyping*, a process where specifications for a piece of software are developed by interaction between a *software developer*, a *client*, and a *prototype program*. Rapid prototyping is used when a client cannot initially define the requirements for a piece of software to a degree necessary to satisfy more traditional design methodologies, such as those defined by Edward Yourdon and Michael Jackson. (p. xi)

There is a sharp distinction between the prototyping approach and traditional methodologies in respect to the assumptions made about *knowledge* in the software development process -- who knows what, when, and in what manner. Traditional methodologies implicitly view the relevant knowledge as articulable and static. M.F. Smith points to three assumptions these methodologies share:

The first assumption is that all the requirements and needs of applications can be analysed and understood adequately by the users and software developers before development begins.... There is also an assumption ... that software needs and

---

<sup>10</sup> Joint application design (JAD) and rapid application development (RAD) are related approaches. Joint application design stresses bringing together all the people who should make a contribution to the design; rapid application development involves a combination of joint application design sessions, CASE tools, and prototyping. For our purposes, what is crucial to all these is the interactive learning at which they aim.

<sup>11</sup> This does not mean the customer should have carte blanche to ask for new features. Rather, it is a means of discovering and verifying what the customer needs.

requirements will be stable.... Finally, there is an assumption ... that users understand fully the technical documentation presented to them. (1991, p. 4)

The prototyping approach, by contrast, recognizes that the necessary knowledge is far more elusive, changing, and difficult to communicate. Perhaps most importantly, the clients, for whom the software tool is being designed, do not know what they want, or at least they are unable to say what that is. Much of the users' knowledge is tacit, inarticulate. Accordingly the most fundamental kind of knowledge necessary to the process -- what the software is to do -- is not readily accessible at the outset. As Mullin puts it,

Unfortunately, clients rarely have this complete a grasp on their problem; they usually assume their responsibilities are simpler, namely, they:

- Recognize that a problem exists
- Find an expert to solve the problem (1990, p. xi)

Just what the problem is, nobody is clear about. But if the clients cannot say what sort of tool they want built, how are the designers to find out? Prototyping provides a means.

Prototyping is an iterative process that accommodates adjustment and change; it anticipates instability of requirements, due both to new insights gained by both clients and developers, and to changes in the business environment:

DC 3, p. 57

Requirements and software actually evolve together throughout the lifecycle of the project.... In the iterative approach to software development, users "stay in the loop," refining their requirements as they better understand what application features are possible... (Adams 1992b, p. 7)

The prototype itself serves as a valuable communication medium through which designers and users can reach reasonable confidence that they really understand one another. Because the process is iterative, it allows for frequent, regular interaction. Because the prototype is a version of the evolving product, the dialogue has a clear, mutually understandable focus. Instead of having to make sense of a lengthy requirements document and figure out if that written description really captures what they want (or think they want), users can interact directly with the prototype and experience whether or not it meets or fails to meet their needs.

CR 5, p. 53

*It is perhaps understandable that traditional methodologies should assume well-understood, fixed requirements: the computer field is very young, and many early programs were essentially electronic replications of existing manual systems such as inventory management and accounts payable. In these kinds of cases, the knowledge of what the tool must do is mostly available. The users know pretty well what they want and express it reasonably clearly. The software designers have the added help of being able to look at what is being done on paper. In such circumstances, it was not*

*so necessary for developers and users to carry on a dialogue through which they could come to understand one another.*

*But the old methodologies are severely strained under present conditions. Today,*

*[s]oftware developers are no longer confronting situations where they are reproducing manual systems. Now they are expected to replace a chunk of the client's middle management with an expert system, one that uses all of the system's existing data to decide such things as when to reorder, how much to reorder, what bills to pay, and what customers are good credit risks.*

*...[The designer may sometimes] be lucky enough to get a clear definition of the problem and be able to see an immediate solution. ... More often, a client will say something like, "Gee, this system has completely changed the way we do business. And now we have all of these great ideas about how we can get the system to do even more for us." Unfortunately they can't give you a lot of detail about these new ideas. After all, that's why they hired you. (Mullin 1990, pp. 2-3)*

A new software system's requirements cannot be fully known, and hence the software's capabilities cannot be fully specified, at the project's inception. In this lies the problem with traditional methodologies based on the classical "waterfall" model, in which design begins after the software requirements are (supposedly) fully specified and analyzed.

[T]he conventional 'waterfall' methodology practiced in most large companies today ... requires the creation and approval of numerous detailed documents before the first procedure is ever written ... [and] doesn't allow any modifications once the actual programming has begun. This constraint frustrates [client] managers to no end because they rarely know what they really want until they see it running on a screen, at which point it's too late to make any changes! (Taylor 1990, p. 97)

The difficulty of this approach is illustrated in the experience of one developer working on a project that was to provide "the usual project deliverables of requirements specification, functional specification and design specification which cover the specification phase of a development project." They found that

[t]he functional specification standard ... was too inflexible for the needs of the GUI [the graphical user interface they were building]. Other techniques such as formal specification were inappropriate considering the time constraints.

Thus a more pragmatic approach was accepted -- that of prototyping. (Barn 1992, p. 25.)

The point of rapid prototyping is to establish the requirements, to find out what the tool must do. "Your job as a rapid prototyper" says Mullin, "is to work with the client to extract specifications

for their new software." Early in the process, your focus "is simply on defining *why* this software is being written in the first place, which will tell you *what* is expected of it." (1990, pp. 214-215)

Requirements elicitation is a basic purpose of rapid prototyping, which takes a wholly different approach to software development from that of traditional methodologies. Simply put, rapid prototyping works as follows: After an initial meeting between client and developer, the developer produces a very simple prototype which the client can try out on the computer. Then follows a repeated sequence of the following steps:

- the clients try out the current version of the prototype and react to it. They explain as well as they can what they like and don't like. Equally important, the developers observe what they do and don't do, what they try, what they ignore, where they are frustrated, and where they are pleased. CR 5, p. 53
- informed with this new knowledge, the developer improves and extends the prototype, and offers this next version to the client for trial.

The cycle continues in a kind of dialogue -- a conversation in which the prototype itself is passed back and forth, as much as any words about it -- until the prototype has been refined to where it contains the functionality the client needs. At that point the initial version of the software to be delivered is defined, and the developers' emphasis turns to details of implementation. (This transition is a change in emphasis, rather than a switch from one set of activities to a distinctly different set. Analysis, design, and implementation are all really occurring together throughout the software development process.)

Note how prototyping can help bring out important knowledge that is initially hidden or non-existent (as yet). Prototyping brings out *latent* knowledge. Some of those whose knowledge needs to be incorporated in the software will not be consciously aware of their knowledge. That knowledge will need to be brought out in application to a problem. The dialogical process of prototyping serves to trigger the re-discovery or creation of useful knowledge on the part of the participants. The users' reactions stimulate the design knowledge of the designer, and the functionality offered in successive versions of the prototype stimulates the users' knowledge of function, helping them become more clear as to what they want the software to do. Helpfully, the prototype itself provides the medium in which these different kinds of knowledge may be captured. CR 3, p. 52

Prototyping also serves to draw out the tacit knowledge of both clients and developers. This is the knowledge they have, but cannot articulate. Even when clients seem to know at some level what they want, often they cannot express it. The prototyper can discover what they know by showing them different capabilities and carefully attending to their responses. Significantly, however, it is not only clients who have knowledge they cannot articulate: much of the designers' knowledge is tacit as well. They cannot say precisely what makes good design, nor why they take some steps rather than others. Prototyping provides software designers a process in which

to draw from themselves their own latent knowledge of how different kinds of challenges should be approached, without committing too early to design decisions difficult to change later.

In addition to being dispersed and tacit, the knowledge valuable to a software development project is generally *incomplete*. It accumulates continually over time. This is why prototyping must be iterative. Every time designers listen to feedback from their clients, their knowledge of the clients' wants increases, and every time the clients interact with a prototype, their knowledge of the software's potential increases. In this manner valuable knowledge accumulates.

### **Software development as interactive learning**

Because the different knowledge that must ultimately be incorporated in a new software application is dispersed, tacit, and incomplete, the development of new software capital is necessarily a discovery procedure, an *interactive* learning process. Through the interaction of all those who have the knowledge the software needs, that dispersed knowledge is brought together in the new software. The knowledge gets built into, coalesces in, becomes embodied in, the software. Developers should tune their development processes to foster this valuable interaction.

In an important sense, it is actually the new software itself that "learns." The client never learns what the designer knows of modularity and information hiding; the designer never fully understands the client's management style, to which he is tailoring the system; the programmers never learn why the screens must look like *this* instead of like *that*. The only "place" in which all the relevant knowledge truly resides is the software itself.<sup>12</sup>

*On this view, the development of new capital goods can be seen as a prime instance of the social cooperation of the market process. Just as the farmer, miller and baker cooperate in producing bread for others to consume, so the client, designer, and programmer cooperate in producing new software tools for the client (and others) to use in further production. The knowledge inputs of all are necessary, and the only "place" where they exist together is in the bread or the software. Anyone who eats the bread or uses the software thereby takes advantage of the knowledge contributions of all those who have participated in its production.*

The learning process of software development is non-deterministic and evolutionary; it cannot be automated, and it defies capture in a rigid methodology. As Mullin says,

I have stressed that modern software development often has little resemblance to the formal development process taught in schools and industry accepted texts. Instead, it's much more of a hit-or-miss affair, with everyone stumbling around in the

---

<sup>12</sup> For an intriguing explication of the complex interdependencies of our knowledge, and the degree to which we unknowingly draw on a tremendous amount of shared knowledge and understanding, in our routine activities, see Phil Salin's article on "The Wealth of Kitchens." (1990)

dark, hoping that they will trip over the correct solution to the problems confronting them. This arises primarily from the fact that software development is innately a human process, as opposed to the mechanistic process many claim it to be. If such an argument were true there wouldn't be much need for programmers, as our current technology is well suited for automating mechanical tasks. When the task requires creativity and insight, our technology is of little use. (1990, p. 136)

Mullin overstates here. It is not that our technology is of little use, but that we must use it differently when we are learning than when we are mechanically applying what we have learned.

Let us look more closely at the nature of this learning process as illustrated by prototyping. Inevitably the process is interactive, because the relevant knowledge of function and design are dispersed and must be brought together.

### **Interaction between user and designer**

In a discussion of a product they built for Hewlett-Packard, Bob Whitefield and Ken Auer of Knowledge Systems Corp. bring out the inescapable necessity of interaction between client and designer. The product is called the Hierarchical Process Modeling System (HPMS); it provides computer automation for Hierarchical Process Modeling (HPM), Hewlett-Packard's means of modeling its internal business and manufacturing processes. Whitefield and Auer explain that they rejected one development possibility because "the development time and costs were prohibitive considering the immature state of the HPM methodology. What was needed was a quick and inexpensive prototype *to continue exploring what kind of tool HP really needed*" (1991, p. 65, emphasis added). Because Hewlett-Packard was still developing HPM, clearly they were unable to define it fully for Knowledge Systems. The methodology and the computer tool which was to represent it were to co-evolve in an exploratory process of interaction between client and software designer.

CR 3-7, p. 52

As a specific illustration of this interaction, consider the following excerpt from Whitefield and Auer's description:

In addition to its graphical representation, each component also has a semantic counterpart. It is entirely possible to create and edit models using only textual browsers, but few users ever do so. In fact, users spend so much of their time using the construction diagram that they tend to think of the diagram itself as the model. As the key nature of the construction diagram became apparent, the following requirements were established for the final tool: ... (p. 67)

"Became apparent" is the revealing phrase here. The users of HPMS at Hewlett-Packard did not specify at the outset that for their purposes, a picture was worth a thousand words. The designers learned this through interaction with their clients. Without this interaction -- suppose, for instance, Hewlett-Packard and Knowledge Systems had tried to proceed by traditional "waterfall" methodology and begin with a document containing all the software specifications -- the

knowledge would probably not have emerged, or at least not without a great deal of frustration, misunderstanding, and delay.

### **Interaction between user and evolving design**

Note that there is another kind of interaction at work here: that between the client and the evolving application itself. The reason the HPMS users did not specify the importance of the diagrams is probably that they themselves did not realize it; after all, they had never used this kind of tool. The users discovered what they wanted and needed through interaction with the application itself, as it evolved.

Whitefield and Auer are explicit about the discovery that occurred as the users interacted with the prototypes. They say, for example, "As the alpha version of HPMS began to be used, response time was determined to be a critical factor in user acceptance. A goal of less than two seconds to route and draw most diagrams was established for the final product..." Also, "HP often desired cosmetic changes to diagrams. *As experience was gained with the tool*, flaws in default placement and appearance were uncovered. This was expected, although the extent and types of changes were not" (p. 67, emphasis added). The users at HP needed to use the tool to realize their speed requirements and to identify the flaws in the defaults. CR 3, p. 52

Two aspects of the discovery process show up in the client users' interaction with the prototype. One has to do with the tacitness of knowledge. Much of the users' knowledge of their work is tacit, inarticulate -- they know what they do much better than they can describe it. Therefore one element of this discovery process consists of the users' discovering in the conscious, articulate part of their minds the knowledge that was there tacitly. In using the HPMS prototypes, for example, the users at HP bring their tacit knowledge to bear, and where the tool does not match smoothly with what they actually do, they detect problems. Of course they need not be able to explain these problems completely; they need simply say what they don't like and what they would prefer, clearly enough so that the designer can improve those areas in the next version of the prototype. CR 5, p. 53

Another, more subtle aspect of the discovery process has to do with the incompleteness of knowledge. We have examined the bringing to light of knowledge which already existed, but not in communicable form. In some sense the users of a prototype know that they want and need certain capabilities in the software, but they are unable to express these needs to the designers. Now, by contrast, we consider the discovery of capabilities that the users do *not* want or need at the outset, because those capabilities never occur to them in any manner. Only in working with the prototype do they first conceive of these capabilities. Once they do conceive of them, however, they want them.

The working prototype provides a context in which previously unimagined possibilities can come to mind. One programmer and tester of new software says, "When I try out a new user interface, I find myself trying to do things with it. When it won't let me, I'm frustrated." The interface --

what the users see of the prototype -- suggests possibilities to the users. It provides them with a new way to look at what they do, and this look may generate new insights as to what they *might* do.

### **Interaction between designer and evolving design**

It is not only the users who interact with the evolving application, of course; the designers do also. This may seem so obvious as not to need mention: how could the designers ever produce their designs without interacting with them? The point to be stressed, however, concerns the nature of this interaction: the designers themselves are engaged in an evolutionary sub-process of generating new knowledge which shapes the evolving design. They are learning also, not just about what the client needs, but about what they themselves can do. Ward Cunningham, a widely-respected programmer, designer, and methodologist, describes some of his design experience in these terms:

We'd get an idea, type it in, and say "Let's see what that does." Kent would ask me a question. I would say, "I don't know," but I'd just start typing and we'd let the machine tell us.<sup>13</sup>

Designers' knowledge of design principles and various problem-solving techniques is not all ready to hand, nor is it static and complete. They discover how to apply this knowledge to new problems in the process of applying it. They ponder, they sketch, they experiment, they try out various ways of decomposing the problem, they make some initial decisions, they repeat the process. They learn by doing.

*A good illustration of the manner in which the designer learns through working with the design comes in Mullin's description of the initial laying out of the views (screens) that the user will see:*

*The actual act of laying out the view provides you with another set of information you will need in constructing the prototype. By deciding on the visual grouping of information in the view, you will also be determining any data assembly, or aggregation, capabilities that the view needs.*

*By laying out a view, you learn something; by deciding on grouping, you determine needed capabilities. In brief, by interacting with the evolving design, the designer learns more about what it should be.*

A creative process such as software design is not deterministic, with output dictated by input through some sort of black-box optimization. This would require the designer to grasp the problem in its entirety at a glance, and on that basis to grasp its "correct" solution. On the contrary, software design is an evolutionary process in which the designer "makes sense" of the

---

<sup>13</sup> Personal interview conducted October 1992. "Kent" is Kent Beck, a respected Smalltalk consultant and president of First Class Software.

problem over time, and gradually puts the design together. In this respect software design is akin to writing. Composition is not a matter of copying out a book that has somehow popped into the writer's head. Rather the writer works gradually from a vague idea to a fully-conceived book, through a process of fleshing out, defining and refining, finding out what "works" by trial and error. Similarly the software designer uses feedback from the design itself, seeing what works, what has promise, what relationships are revealed that were unclear before.

DC 7, p. 58

### **Iteration: the design dialogue**

We have discussed interaction between client and designer, and interaction between both of these and the prototype itself. These two kinds of interaction are closely related in practice, even the same in a sense, because it is largely by means of their interaction with the prototype that the two groups interact with each other. The prototype is a communication medium. Those involved communicate with one another largely in their responses to the prototype, with these responses closely observed by the other side. In a sense, there is a dialogue going on in which the prototype is passed back and forth. The designers say, "Give this a try," and watch. The users try it out, experiment, exclaim about some features, pout about others, ask questions and describe frustrations. "Well, this part is good," they say, "but that part needs to be more like so. Set up as it is, I can't do such and such." The designers, in turn, think, "So *that's* what they want! Why didn't they say so in the first place? Well, I can give them something twice as good as what they're asking for. Wait until they see this..." In the next iteration, the user may respond, "No, no! That's not what I need! But it's marvelous! You can do that?! Well then do it this way...!"

This fanciful example illustrates another important characteristic of the learning process that is software development: it is *iterative*. Both sides in the dialogue are learning from one another. On the basis of what they learn in each round of the exchange, they change what they feed back to the other side, thereby calling forth new learning there. The process is gradual because learning takes time. The new software develops throughout this ongoing exchange, as more and more of the necessary and appropriate knowledge gets built into it, and extraneous, unnecessary knowledge is discarded. Here is Mullin, again:

CR 5, p. 53

In RP [rapid-prototyping] design, we stop designing on a regular basis in order to run the prototype by the clients and users, thereby getting information on adjusting our design before it's too late to do anything about the parts they hate, or the things they really wish it had. So, for all my arguments about seamless development environments, it appears that our design actually progresses by fits and starts, as opposed to the seamless path of traditional design evolution.

As it happens, this observation is wrong. These sessions with the client are not "seams" in the RP design process, they are natural components of it. They provide us with the means to continually adjust our design course and goals as we learn more about what the client desires by letting them interact with our best idea of what it is that they do desire. As they do this, they will provide us with the necessary

information to extend the design another level. Recall that I observed at the outset of this book that it wasn't realistic to expect to get a clear list of requirements from a client when you commence a design project. We are designing to the requirements we have and then using that design to dig up more requirement information (1990, pp. 86-7).

---

## Understanding the evolving design: the value of a rich development environment

### Summary

Because software development is a learning process, developers should choose and tailor their development environments to aid the development team in understanding, thinking about, *and altering* the evolving design. A good development environment will provide developers

- a variety of ways to look at the design,
- rapid feedback from the design, through incremental development capabilities and a good debugger
- excellent change tolerance

We concentrated in the previous section on the social, interpersonal aspects of software development, through which dispersed and often tacit and latent knowledge gets accumulated and expressed in code. In this section we take a different perspective on the same process, concentrating on the learning aspects, on *how* that knowledge gets successfully articulated into working code. We do this through an examination of some of the higher-order tools that help software designers do their work.

*The fundamental challenge in software development is to make sense of the complex systems we are trying to build: to understand them and the way they function, and to express that understanding in code. The primary constraint on programming today is not physical resources, but the very complexity of what developers are trying to do, and the limits of their ability to manage that complexity. As Mark S. Miller of Agorics, Inc. puts it, the key limitation is "our sheer ability to understand what it is we are trying to do."<sup>14</sup> Most of the tools software designers use, higher-level programming languages in particular, are usefully understood as tools for helping them *understand* what they are doing.*

---

<sup>14</sup> Personal conversation.

## ***CASE tools as tools for understanding***

A survey of the main features offered in current CASE tools<sup>15</sup> reveals the following ten basic functions, grouped under four headings:

### diagramming support

1. draw diagrams
2. check diagram consistency

### data management

3. provide requirements database and requirements tracing
4. provide data dictionary
5. provide repository management (for workgroups)
6. support change management and version control

### prototyping support

7. prototype (usually "screen prototyping")
8. paint screens

### code generation

9. provide facilities for porting between platforms
10. generate code

With the exception of the last category, each kind of tool serves to helping designers *learn* about the systems they are building. The diagramming tools produce data flow diagrams, entity relationship diagrams, or program structure diagrams; or they do modeling -- systems requirements modeling, data modeling, behavioral modeling. All of these visualizations are aids to designers' understanding of the complex system they are creating. And of course where a team is doing the development, the diagrams help maintain coordination among the team members, by giving them a shared focus for discussion and a helpful visualization of what others are doing.

The tools for checking diagram consistency provide important feedback to the designers from the evolving design embodied in the diagrams. Automated diagram consistency checkers point out all the places where a version of the design is inconsistent or nonsensical, i.e., where the designers have not fully grasped all the ramifications of their actions. For a example, sometimes designers will specify all the inputs necessary to a particular module, but fail to specify any output. Diagram consistency checkers point out such flaws automatically.

DC 7, p. 58

The data management tools serve primarily to help maintain coordination among the members of a development team. Modern software development is very much a social process, as we have seen, depending on the contributions of many. In this context it is helpful to have a shared database where information about the project can be stored and accessed. Because the requirements of a system develop and change as the system takes shape, it is helpful to have a history of their evolution, so that team members may understand why

DC 9, p. 59

---

<sup>15</sup> This particular listing is drawn from Kara (1992).

something is being done as it is. Repositories are databases where segments of code, modules of the system, can be stored and accessed by different members of the team. And of course as changes are made and different versions of the system are developed, it is important that coordination be maintained to avoid conflicts and inconsistent expectations. In a general way, all this information provided by the various CASE databases serves to help the developers understand what is happening, to grasp the nature of the systems they are developing, so that they may contribute their own knowledge to it.

The value of prototyping we need not repeat: it helps the designers understand the needs of the users and the users understand the operation of the evolving system, so that both may better come to understand what the system can and should be.

All these tools are best understood as tools for learning, for making sense both of what one has done on one's own and of what others on the same team have done and how that affects the whole. This kind of conceptual development is the designer's bread and butter.

If you watch how a designer works you see lots of things going on which give you some insight into the thought processes going on. Sometimes a designer is just trying out some new idea. Sometimes a designer is evaluating or making some catastrophic change to previous ideas (maybe about 90% of the pictures a designer draws get thrown away). Sometimes a designer is trying to customise something developed for another purpose. Sometimes two designers who have developed separate pieces of a solution are trying to bring them together. Sometimes a designer is checking that all of the ideas actually hang together. One thing you will see is that very little time is actually spend on the finished product. (Robinson 1992, p. 4.)

### ***Object-oriented programming environments***

Object-oriented programming environments such as Smalltalk provide some additional tools not included in the above list of standard CASE functionality. Some of these tools are especially useful to software designers in the *learning* process that is software engineering.

#### **Incremental compiling**

One of the most useful and important aspects of Smalltalk is that any chunk of code, no matter how small, can be run -- and produce meaningful results -- at any time. "Smalltalk is an incremental environment. Small, incremental changes are small efforts."<sup>16</sup> This incremental compiling capability is in marked contrast to earlier programming languages, in which the whole program has to be complete and accurate before it can be run. The importance of incremental compiling to learning has to do with the complexity of software -- we might think we know what a

---

<sup>16</sup> Ward Cunningham, personal interview, October 1992.

piece of code does and how it interacts with other pieces, but often we don't. In developing a system, it is extremely useful to check in with reality at regular intervals, to make sure we understand. The ability to run each module of Smalltalk and look at the results gives programmers the benefit of rapid feedback from the system; it allows them to understand it better and sooner. As one programmer puts it, "your thought processes don't get interrupted; you don't leave the context."<sup>17</sup> Additionally, incremental compiling leads to higher quality, because smaller chunks of code are easier to test. DC 8, p. 58

### Step-by-step debugger

A related capability of Smalltalk is a built-in debugger. This is a tool for tracing exactly what happens, step by step, so that when an error occurs or something unexpected happens, the programmer can find the cause of the problem easily. This capability provides rapid feedback that is immensely valuable to programmers' understanding of the system. Ward Cunningham credits this feature with a large part of the reason why Smalltalk is such a good development environment:

There was never a risk of a bad bug, because whenever something went wrong, we'd get a notifier [debugger], hop in the notifier, and it would tell us what went wrong. We were never in a position where we didn't know the next thing to do to diagnose our programs.<sup>18</sup>

It is important that the ongoing interaction between the designer and the design not be interrupted too long. Less-capable languages cannot tell a programmer where something went wrong, only that it did. In these circumstances it is possible to be absolutely stumped. When that happens, the programmer then has to search for the problem. In so doing, he loses the context; his thought processes get interrupted. Additionally, the whole program usually has to be recompiled and rerun before he can make sure that he has fixed the problem correctly. The sheer time this all takes is distracting; it makes it difficult for the programmer to concentrate on solving the problem before him. DC 8, p. 58

The combination in Smalltalk of incremental compiling and the built-in debugger is especially powerful. When one "hits a bug" in Smalltalk, a debug window appears in which one can usually fix the problem quickly and easily. This change to the program is automatically and immediately compiled and linked into the rest of the program. Accordingly, it is not necessary to go back to the beginning, recompile and begin the program again. Instead, one can simply continue with the

---

<sup>17</sup> Lee Griffin of IBM, personal interview, October 1992.

<sup>18</sup> Personal interview, October 1992.

program in its newly repaired state, by clicking the "Restart" button on the debug window. Smalltalk users find this feature extremely important.<sup>19</sup>

Of course bugs occur in all programming, but most programming languages are ill-equipped to help developers deal with them. Traditional programming languages rely on the program's being correct; they assume that the end user is the only person whose efficiency needs to be optimized, and aim to give the end user the fastest possible program.

Smalltalk, by contrast, recognizes in its very design that we live in a world of error. The designers of Smalltalk took seriously the learning challenges of designing software, and therefore provided incremental compiling and a powerful debugger to support software developers.<sup>20</sup> One developer enthusiastic about object-oriented languages says, "these languages talk back to you and let you know when you are doing a good job." The difference between "Smalltalk and C++ is that Smalltalk talks sooner and louder when you are doing a bad job."<sup>21</sup>

While I confess to being a Smalltalk lover, both from my own experience and as a result of my research, the point here is not to push Smalltalk. The point is to push the kinds of capabilities that Smalltalk provides for giving software designers immediate and detailed feedback from the systems they are building. Such feedback is extremely valuable to learning about the system, and learning is the essence of software development. Developers should choose environments with this kind of capability unless they have extremely good offsetting reasons for doing otherwise.

### **Multiple views into the system**

Smalltalk also provides tools that give users a variety of different perspectives on the code. For example, there are hierarchically structured browsers for viewing the different elements of the code in the system. In addition to providing a handy means of looking up and accessing some particular class of code, browsers significantly aid understanding of software systems by providing a meaningful view of the relationships between different elements of the system. *Where* a piece of code is located in a browser window often carries more information than the details of the code itself. Smalltalk also provides windows which display relationships between objects, such as which kinds of objects send messages to others. It also provides windows for viewing the actual values of variables pertaining to particular elements of a system.

These different views into a complex system are very helpful in understanding it. These views give designers different perspectives into software systems, making various relationships more understandable. A complex system, by its nature, cannot be wholly understood by a particular person at one time. But it can be understood better and better in proportion that one has a variety

DC 7, p. 58

---

<sup>19</sup> Richard Collum, systems developer in a large Smalltalk product at First Union National Bank of North Carolina, says simply, "The restart button is the greatest thing." Personal conversation, October 1992.

<sup>20</sup> I am indebted Ward Cunningham for explaining this distinction.

<sup>21</sup> Paul Ambrose, personal telephone conversation.

of different perspectives on it. Each new perspective enriches one's understanding of the other perspectives, and hence of the system as a whole.

*On this point, Mark S. Miller has said that he has reservations about tools that generate code from diagrams. He prefers tools with which*

*you write the code and have it generate the diagrams. That's superior because whatever you are programming in has to express the entirety of the program, and people have found words and symbols to be superior for that purpose. But the visualizing tools do a good job of representing a slice of or aspect of the program, with different tools providing different slices.<sup>22</sup>*

---

## Communicating about the design: the value of code that is similar to natural language

### Summary

Because software development is a *social* learning process, developers should choose development environments and programming languages that will help them and their customers communicate effectively about the evolving design. For this purpose, the more similar the language to natural language the better.

One of the advantages of pure object-oriented languages such as Smalltalk is that they let software developers design and implement with a terminology that is suitable for thinking about the problems they are trying to solve. The terminology, it is said, maintains a "proximity to the problem space." Hence these languages, and the development methodologies built around them, are not just tools for expression, but tools for *thinking* and *learning* about complex systems.

Bertrand Meyer, a leading theorist of object-oriented technology and author of the object-oriented language Eiffel, points out that traditional languages are hard to read and understand. When we look at their code, the relationships are not clear to us. This, he surmises, helps explain why diagrams are so much a part of the structured analysis and design methodologies used with non-object-oriented languages. "[A]fter all," he says,

if you are programming in BASIC or C++ you do need higher-level tools and notations if you ever hope to explain or just understand what is going on. But ...[w]ith object-oriented techniques, implementation becomes high level enough to cover what was traditionally covered by design or even analysis. The same notation may be applied throughout, at various levels of detail. For analysis and design, high-level facilities such as classes ... provide the key descriptive and structuring facilities. For the final implementation, classes obtained earlier are completed with

---

<sup>22</sup> Personal telephone conversation.

the details of the algorithms and data structure implementations. (Meyer 1991, p. 39)

"The same notation may be applied throughout" the development process, from high-level tasks such as analysis and design through to low-level implementation, for two reasons. First, object-oriented languages, like other high-level languages, allow us to specify things in ways more removed from the concerns of the machine -- at a higher level of abstraction. But object-oriented languages are additionally significant but because they let us create our own vocabulary. In these languages we can tailor the terms of the code to the problem space as we understand it, both for thinking about the problem *and* for implementing a solution to it.

DC 10, p. 59

Programming the solution to a problem in a language like Smalltalk is a matter of creating objects and methods which represent, respectively, the entities we wish to model and their behavior. Meyer says

This is the seamless property of O-O development, which yields some of the major advantages of the approach -- among others, the fact that the results of analysis and design are not lost or recorded in some obscure intermediate documents or diagrams, but fully embedded in the final delivered software. (Meyer 1991, p. 39)

Building new capital goods is a matter of embodying knowledge in a usable form: object-oriented languages are effective tools for this embodiment because the terms in which they let us embody our knowledge are so similar to the terms in which we naturally develop and express that knowledge. Object-oriented languages provide software designers more immediate access to the problems they are confronting. In using terms with immediate relevance to the problem domain, they avoid loss of meaning in translation. They shorten the conceptual distance between the knowledge that goes into the software and the software itself. In much traditional structured analysis and design, the designers do their thinking with diagrams, which then must be converted into code by some translation process. Object-oriented languages, by contrast, allow the designers to think in understandable code, thereby providing them a more immediate grasp of the system, and obviating much of the need for translation.

Object-oriented languages, then, help us to bridge the semantic gap between analysis, design, and implementation. There is no semantic gap, because the semantics are the same throughout. In this respect, object-oriented languages are superior tools for thinking about -- and therefore learning about -- complex systems.

An additional benefit of programming in a language that is close to natural language is better communication between developers and their clients. Knowledge Systems Corp., a major development consulting firm specializing in Smalltalk,

found, by modeling entities in terms of their behavior and interaction, that both internal software objects and external entities can be represented in such natural

ways as to be accessible to non-computer professionals like users and domain experts. (Adams 1992a, p. 5)

This approach takes the idea of software development as a social learning process to its fullest extent. In order to begin developing the classes that will eventually be used in prototypes and evolved into a complete, running system, the software designers at Knowledge Systems Corp. use role playing. The process is overtly social, in that various different people with different knowledge and skills are involved on the spot. It is overtly a learning process in that it is a trial-and-error method of discovering what the important objects in the software should be, and how - by what methods -- they should interact with one another. Sam Adams describes their experience:

While most methodologies rely on diagramming notations to attempt to capture and communicate complex interactions between objects, role-playing allows the designers to actually experience the behavior firsthand. This theatrical anthropomorphism has many benefits in the design process. Since designs can be "executed" very early in the process using scenarios, alternative designs can be explored easily using role-playing as a form of rapid prototyping. Designs as complex as entire manufacturing systems can be simulated in surprising detail, taking advantage of the temporal and spatial nature of role-playing that can be only poorly captured on paper... An additional benefit of role-playing in design groups is that it tends to help involve everyone in the design process, regardless of their background or experience, so all participants can add their unique value to the process (1992a, p. 6.).

We are accustomed to think of a programming languages and computer program as vehicles for communication between people and computers, the means by which people tell computers what to do. But because software development is so much a social activity, languages and programs need to be equally -- perhaps more so -- vehicles for communication among people. Even when programming is in fact a solitary activity, when one developer writes an entire program herself, there is great value in using higher-level, more expressive languages closer to natural languages, because these help the solitary programmer think about the problem and understand what she is doing as she works. The language helps her communicate clearly with herself as she works.

When programming is done in teams, and when the program must evolve over time, the importance of clear communication is multiplied. Kent Beck writes that "[t]he cost of a piece of code over its many-year life is dominated by how well it communicates to others.... Every method name, every class name is an opportunity for you to communicate what is happening" (1995, p. 20). Effective software development depends on effective communication among various developers and among the developers and their clients. Hence those languages which best foster interpersonal communication best foster effective development.

DC 11, p. 59

---

## Coping with constant change: the value of modular design

### Summary

Because software development is a learning process, insights as to what the design should be will accumulate throughout the design process. Because the ecosystem of production in which software is used is constantly changing, the software must be adapted over time in order to maintain its value. Furthermore, developers sometimes perceive that an existing product can be adapted for new purposes. For all three reasons, changes to an application's design, sometimes even to its fundamental architecture, will often be desired. Accordingly, developers should craft their designs to be as change tolerant as possible. All designs should be modular and object-oriented.

### ***Evolvability as a design goal***

There is general agreement in the software industry that ease of maintenance is crucial. Practitioners in the software world clearly expect continuous change, though they cannot know just what those changes will be.

*In economists terms, they face uncertainty. (Knight 1971) They are foresighted, though without clear vision of the future.<sup>23</sup> Accordingly, they must plan as best they can to meet those changes, whatever they may be. As Hayek says,*

*With respect to [changes of technical knowledge or invention] the idea of foresight evidently presents some difficulty, since an invention which has been foreseen in all details would not be an invention. All we can here assume is that people anticipate that the process used now will at some definite date be superseded by some new process not yet known in detail. (1935, p. 97)*

Software developers have not always anticipated that change would come as soon as it generally does. As we have seen, in earlier days many software developers seem to have overlooked the pervasiveness of change, and tried to build software to specifications they assumed to be fixed. But the years and budget overruns have made the lesson painfully clear: DC 4, p. 57 change never ceases. Indeed, it seems to accelerate. Accordingly it makes sense to build software so as to facilitate change in general. Good design, in an uncertain world, is design which prepares for change. The more easily, quickly, and inexpensively software may be

---

<sup>23</sup> Lachmann writes:

[T]he purpose of all capital, hence also of the current maintenance of existing capital goods, is to secure a future income stream. But the future is unknowable, though not unimaginable, and men have to use knowledge substitutes in order to evaluate future income streams, viz. expectations. (1975, p. 2)

maintained, revised, and adapted for new purposes, the more valuable that software will be over time. A major goal of good software design, then, is to ensure design *evolvability*.

### **Co-evolutionary development**

The evolution of complex systems such as the ecosystem of production is not a movement toward some particular endpoint, or even in some particular direction. Evolution is necessarily *coevolution* of the different elements of the system. In the capital structure, this means that which tools become useful and which become obsolete at any time is determined by what *other* tools happen to be developed also, and what other technologies happen to be discovered.<sup>24</sup>

Consequently "the best solution" to a particular problem is a mirage that appears when one focuses on the moment. In another moment the problem will have changed, and there will be a new "best solution" for the simple reason that others have been working on related problems. There is no fixed skeleton or underlying architecture for the capital structure. The skeleton, the architecture, grows organically as particular entrepreneurs make particular choices. Each choice in response to a particular aspect of a problem poses a new, or at least a changed, problem for other participants in the process. In the words of Peter Allen, a specialist on evolutionary dynamics at the International Ecotechnology Research Centre:

Evolution is not just about the solving of optimization problems, but also about the optimization problems *posed* to other populations. It is the emergence of selfconsistent 'sets' of populations, both posing and solving the problems and opportunities of their mutual existence that characterizes evolutionary dynamics (1990, p. 25).

Software developers, then, must try to build their products *so that they can readily evolve* to maintain a reasonably good fit in the evolving ecosystem of production around them, regardless of how -- out of a broad continuum of possibilities -- that ecosystem may evolve.

### **The optimization trap**

Crucially, this means that *optimization* of software for any task *as defined at a particular moment*, should usually be sacrificed for greater flexibility of design. This is not to say that achieving an excellent fit between software and given task should be ignored; of course suitability to a particular set of specifications is important. But hard experience has shown optimization as such to be highly problematical, because optimization trades off against flexibility. As Bertrand Meyer puts it, in discussing tradeoffs among different goals of software design,

---

<sup>24</sup> Other factors include people's expectations, the interest rate, availability of skilled personnel, etc. See Lachmann (1986) and Hayek (1935).

...optimal efficiency would require perfect adaptation to a particular hardware and software environment, which is the opposite of portability, and perfect adaptation to a particular specification, whereas extendibility and reusability<sup>25</sup> push towards solving problems more general than the one initially given (1988, p.7).

Software designs, in today's business environment, are like organisms in an ever-changing ecosystem: if they cannot mutate with reasonable ease, the species is likely to disappear. In this we find an illustration of a basic principle of evolution. In Peter Allen's words,

...evolution does not lead to individuals with optimal behavior, but to diverse populations with the resulting *ability to learn*. The real world is not only about efficient performance but also the capacity to adapt. What is found is that *variability* at the microscopic level, individual diversity, is part of evolutionary strategy... In other words, in the shifting landscape of a world in continuous evolution, the *ability to climb*<sup>26</sup> is perhaps what counts, and what we see as a result of evolution are not populations with "optimal behavior" at each instant, but rather actors that can learn! (1990, p. 15)

In other words, to be successful over time, the entities that populate complex, dynamic systems -- whether species in the natural world or software systems in the capital structure -- must not be optimized for a certain set of conditions, but *evolved for evolvability*. In the software setting, the "actors" are software product versions and lines, which compete in the economy for wider use. A product perfectly adapted for, say, an IBM mainframe system using identical terminals all at one site is likely to be in trouble when the company using it decides to downsize to a network of varied workstations and PCs, communicating over a network spread across five cities. That species of software would be much more survivable and valuable were it less optimized and more evolvable.

DC 5, p. 58

### ***Aspects of software evolvability***

There are two main kinds of software evolvability for us to consider. In Bertrand Meyer's terminology, these are as follows:

- **extendibility** - the ease with which software products may be adapted to changes of specifications, and
- **compatibility** - the ease with which software products may be combined with others. (1988, pp. 5-6.)

---

<sup>25</sup> Extendibility and reusability are discussed in the appendix.

<sup>26</sup> "Climbing" here refers to "hill-climbing," a metaphorical term in ecology referring to the ability of a species to develop characteristics that enable it to flourish -- to climb the "hill," defined in characteristic-space, of characteristics suited to survival in a given configuration of populations and resources.

All software, except for very simple, short programs, comprises *systems* of related functionality. Software is more like a factory, embodying a variety of machines and processes all working together, than like a single machine. In adapting software to changes in the specifications, some elements of its functionality are eliminated, some replaced, and others added. This is similar to retooling a factory to new production demands by eliminating, replacing, or adding machines or processes. Software extendibility is the ease with which these changes can be made.

Similarly, software compatibility is matter of capital *complementarity*. A software application or component is compatible with others when it will work together with them. It is incompatible when it is so highly specialized for some particular original purpose or context that it cannot easily be made to work with the others.

### **Sources of change**

There are at least four categories of reasons for change (apart from the obvious change needed due to bugs) with which developers should be concerned:

1. Change necessitated by insights gained in using the software. These insights may be gained during initial development, or after deployment. In the first case, they would require what is called “design changes.” In the second, they would require what is called “maintenance,” or “enhancement.” As is well known, maintenance costs generally constitute more than half of all software costs.
2. Change necessitated by changes in internal or external business needs. Users' requirements may change as their businesses change; the software may need new features to keep up with competitive products; it needs to run on new machines, to be used on networks or intranets, and so on. These changes also may arise either during initial development or after product release.
3. Change to a subsequent version of the same product. This is perhaps a difference only in degree from the first two: When enough new insights are gained from experience with an application, and as the business setting in which it is used changes enough, a new version of the product becomes very useful.
4. Transformation of a product into a related, new product. Frequently it would be profitable to generalize a product developed for some specific purpose for a larger family of purposes. Alternatively, a product developed for one domain might be transformed for use in a related domain, using some abstractions common to both.

Whatever the reason for the changes desired, and whatever the stage in a product's development or life cycle, developers stand to profit in proportion to the change tolerance of their software. The more easily an existing application or any design elements of it can be altered for new purposes, the lower will be the costs of getting out a new or improved product.

### ***Change tolerance through modularity***

I have stressed that software development is a social learning process, and suggested that in an important sense it is the software itself that learns -- the software embodies the knowledge of many contributors, each of whom knows only a little of what the others know. Only in the software itself is all the relevant knowledge to be found. It follows that what it means for software to be maintained -- changed, adapted, enhanced -- is for it to come to embody more and different knowledge than it embodied before.

What are the characteristics that allow software to embody new knowledge readily, that make it change tolerant? These characteristics can be summed up in a single word: *modularity*. Modularity facilitates the social learning process of software development fundamentally by making the software system more understandable to those who must build new knowledge into it.

The importance of modularity to understandability, and hence to change tolerance, can hardly be overemphasized. The principles are however, fairly well known. Accordingly I have placed a long discussion of the issue in an appendix entitled "Change Tolerance Through Modularity."

### ***General comment on modularity and social learning***

Let me emphasize a major point: Modularity is not important to the software system itself, considered as a tool in use. To the compiled and running program, the modularity of the design is utterly irrelevant (as long as the system runs correctly); it has nothing to do with size, speed, efficiency of machine resource use, or other technical matters. But to the program as a design that evolves over time, modularity is all-important. Modularity facilitates the process of software evolution. It serves not the computers it runs on, but the people who develop the system. A more modular system may not run quite as fast as a non-modular system, and the first version may not be delivered to the customer as fast as if the development team ignores modularity and cobbles the system together as fast as they can. In the long run, however -- indeed, in anything but the short run -- it is far more important to have a system that can evolve. Elegantly modular systems are easier for *people* to understand and work with, and *that* makes them more evolvable.

In the on-going process of software development, system modularity helps people deal with the complexity of the systems they are evolving by making those systems more understandable, and by reducing the amount each person needs to know in order to contribute.

The principles of modular software construction are not easy to achieve and sustain. Because there is always a temptation to hack a quick solution rather than maintain sound modularity, it requires constant thought and work to adhere to these principles, to keep a program evolvable as it evolves. Ward Cunningham says that in order to control complexity, "when you learn something about how you should have done it, you have to change the program to do it the way you should

have done it."<sup>27</sup> This is a process he calls *consolidation*, which he likens to paying off the principle of a debt.

Whenever one allows a design to become sloppy, as programmers often do in experimenting with different solutions, it is as if they have borrowed money. The "debt" they owe is the work they will have to do rework the design into clean, evolvable condition again. The main problem with this situation is that the sloppy solution itself leads to problems that are typically addressed with other quick fixes that make the design still *less* evolvable and more brittle. In this way, the "interest compounds," building up the size of the maintenance debt. Eventually, better sooner rather than later, designers must pay off this debt by cleaning up the sloppiness and restoring the modularity of the system, *if* the system is to remain evolvable.<sup>28</sup> What this kind of continual design polishing accomplishes is an appropriate embodiment of the problem knowledge currently available, in a robust, evolvable design. On that design new knowledge may then be readily built. Referring to his experience as designer of the a large financial portfolio management system, Cunningham says that the consolidation process would make

the organization of the program closer to our current thinking. And once we did that we were *free to advance to our next stage of thinking*, instead of being tied back to thinking in terms of the old program.<sup>29</sup>

The point: modularity helps designers understand their designs and this understanding allows the design to be advanced.

DC 3-4, p. 57

---

## Using the division of knowledge: the value of working capital -- components, frameworks, templates, and design patterns

### **Summary**

Because software development is a social learning process through which the knowledge of many becomes embodied in new applications, time and effort can be saved by using the knowledge of others that has already been embodied in software -- in components, frameworks, templates, design patterns, and domain architectures. Developers should embrace a software development process that makes the maximum possible use of existing software. To this end, they should actively invest in working software capital of this type, treating it as a valued corporate asset.

Where software development as currently practiced is most different from the development of hard tools, where software engineering is farthest behind other engineering disciplines and has the

---

<sup>27</sup> Personal interview, October 1992.

<sup>28</sup> Of course, the programming environment itself must be flexible enough to allow such reworking to occur with relative ease. As we have seen, pure object-oriented environments are best at providing this kind of flexibility.

<sup>29</sup> Personal interview, October 1992. Emphasis added.

greatest potential to improve, is in its use of working capital. Too little is used. By the standard of other industries, the software industry has astonishingly little specialization and division of knowledge. In practical terms, this means that developers waste hours and squander profits as their programmers reproduce, on project after project, functionality that has been successfully produced before.

Consider the current state of division of knowledge in most of the software industry: Almost everything except the development tools is built internally. In many cases, programmers literally begin with a blank screen. This is equivalent to a having a house builder begin to build a new house by going out to the forest with a chain saw to cut down trees for timbers and roofing shingles, and digging in the ground to mine iron from which to cast the bathroom fixtures. The contractor may use tools bought from elsewhere, but he produces all his materials himself. To the eye of an economist, long schooled in the value of specialization, division of knowledge, and specialized working capital, this picture is absurd and wasteful. It is a minor tragedy to customers, and a major loss of profits to developers.

### ***Investing in working capital***

Software development need not be so uneconomical. Significant savings in programmer time, as well as improvements in quality and expansion of markets, may be realized from significant investments in working software capital. If developers will embody their knowledge and capabilities in carefully written and well-documented modules and templates that may be used in a wide variety of its projects, they can improve their profitability substantially.

Software components, frameworks, and templates constitute working capital for programmers, to be used in the construction of the software they build. When programmers and designers have functionality already available, they need not build it from scratch; rather they take advantage of the prior work of specialists who have built those inputs for them. Such working-capital inputs to software production are analogous to pre-built motors and gears used by a machine builder in constructing a new machine, or to machines themselves used by a factory designer in laying out a new factory.

The more developers can embody their programmers' knowledge in working capital, ready-to-hand for use, the less new learning is required on any project, and hence the less costly each project can be.

Developers should systematically invest in working capital for their software development. They can do this by buying and/or building software components, frameworks, templates, and design patterns that they can use repeatedly in developing applications. They should do both, in a conscious, continuing effort. The effort should be understood as an investment in working capital that will pay off handsomely over time.

The first, simplest way to build working capital is to buy it. Developers should be on the lookout

WC 5, p. 55

for high-quality, robust functionality they can simply purchase and incorporate as needed.

Additionally, developers should systematically capture functionality that their own development teams develop, that a variety of their customers might use. They should capture this functionality in carefully crafted, robust modules and templates. Development teams should then use these modules in every new project that needs such functionality. Special development teams should be created whose job it is to identify functionality that merits this kind of investment, and to refine, polish and ideally perfect that functionality.

WC 6, p. 56

All company development teams should be trained to use these modules as a matter of course. Every team should have at least one member who is thoroughly familiar with all the functionality available at the company.

DC 2, p. 57

Regarding value, I'm saying here that robust, flexible, nicely abstracted software inputs are very valuable to the developer, because they can be incorporated into new applications at low cost.

## **Templates**

Investing in general templates, which model in the abstract the basic entities and operations of a whole domain, offers great potential payoff. Such templates may concern, for example, a whole industry or a cross-industry business function.

Capturing frequently-used functionality in reusable modules can be made to pay off. But fine-grained, specialized functionality that stands on its own is difficult to track. Often programmers might find it more cost-effective to rebuild rather than use the time necessary to find a small object or component, no matter how elegant and problem-free.

WC 10, p. 57

Large-scale, comprehensive templates are especially valuable. Because they integrate a large body of functionality specifically designed for some particular problem domain, they give development teams at work on particular projects a running start at the job, allowing them to complete rather than build, to tailor a robust basic functionality to the detailed needs of their particular customer.

WC 7, p. 56

## **Benefits of capital-intensive software development**

The ready availability of templates and other software modules will substantially increase developers' rates of new software development and improve the quality they deliver. Systematic investment in working capital should reduce development costs on any particular product (excepting, of course, the early projects in which the working capital itself is developed). It should also increase the range of products that developers can offer cost-effectively: As they carefully capture the essential abstractions of the functionality they build for some projects, they will be able to apply that generalized functionality to other, related projects. For example, if it captures the general functionality needed to manage fleets of buses, it will find that it has captured

functionality needed to manage fleets in general. That functionality can then be marketed to railroad and airline companies for managing those fleets.

### **Saving development time and effort**

The fundamental, obvious benefit of accumulating and using working capital in software development is that functionality built successfully once need not be built again. The savings in programmer time and effort, and the improvements in time to market, are obvious.

### **Freeing programmers to create**

The programmer time and creativity that would be spent reproducing functionality in the absence of a wealth of working capital may instead be spent creating, pushing outward the frontier of the new and challenging. This more concentrated attention on new problems should lead to a more rapid development of new products.

### **Stockpiling expertise**

The range and quality of the templates and other components available to developers should steadily increase with a concerted effort to accumulate this kind of capital. To the extent that this working capital is shared throughout a company, its different teams may “stand on one another's shoulders.”

A number of studies suggest the power of using proven software components to augment productivity.<sup>30</sup> Sam Adams has reported, for example, on a series of products that Knowledge Systems Corp. built for Hewlett-Packard using Smalltalk, beginning with a the Hierarchical Process Modeling System (HPMS) mentioned above. Adams reports that subsequent projects on related systems

benefited greatly from the components developed during the HPMS project. In addition, several of the components were redesigned during their use in other projects and were then reintegrated into HPMS. As a result, several of the components were refined several times across different projects, and became the base for an internal reuse library that has benefited many projects since then. (Adams 1992c, p. 3)

Among the statistics that Knowledge Systems Corp. kept during their work for Hewlett-Packard was an estimate of savings from incorporating existing components in subsequent projects. Adams reports that “[t]he savings often exceeded the actual cost of the project, indicating that much more functionality was delivered for the same cost.”

Note the evolutionary cycle of ongoing development that Adams points out. Components designed in the initial project were then improved in subsequent

WC 9, p. 56

---

<sup>30</sup> See Tirso (1991), Ryan (1991), and Harris (1991).

projects. The new, improved versions were then reincorporated into the first system.

### **Generating economies of scope**

Most programming today still occurs within what Bertrand Meyer calls a *project culture*, in which a specified project "starts at day one with, as its input, some large user's specific need. It ends some months or years later with a solution to that need ..." (1990, p. 76). If developers can move out of the project culture and begin to accumulate working capital for use in a number of related projects, they can achieve significant economies of scope. (Economies of *scale* refer to cost savings that result from producing on large scale, i.e. producing many instances of the same good at low cost per unit. Economies of *scope*, are cost savings that result from producing a number of different, but related products in which the same inputs may be used.)

The availability of working capital inputs which capture the fundamental abstractions of a problem domain greatly simplifies producing a variety of applications within the same problem domain. Templates and frameworks at a high level of abstraction are especially powerful, for these can form the basis of a family of related applications.

High-level templates of this kind can potentially yield tremendous gains. (Of course the gains come at a cost. Finding the appropriate abstractions is challenging. As Sam Adams says, "[t]his level of reuse ... does not come cheap.") High-level frameworks bring forward the starting point at which programmers begin new projects, and facilitate communication and coordination among both the producers and the users of related software products. As frameworks become more generally used, the economies increase.

WC 10, p. 57

### **Cultural shifts required**

#### **Overcoming the "not-invented-here" syndrome:**

Developments organizations' designers and programmers need to overcome their disposition to build for themselves rather than incorporate the work of others. In like manner, they need to build their own code with a conscious eye to that code's use by others, making it clear, understandable, modular, and well documented.

WC 1, p. 55

#### **Eliminating the single-project mindset:**

One of the most important cultural changes needed is a matter of management and business practice. That is, the single-project mindset must be rejected. Software engineers must view what they do as producing not a succession of isolated, independent projects, but a family of related projects with a great deal of common functionality.

Management must support and accommodate this effort. Doing so will probably necessitate a change in accounting practices: The costs of developing generalized, proven software assets must be spread over many projects. Accordingly, the practice of budgeting each project in isolation from others must be given up. Along the same lines, software development contracts must not be

written as they often are today, with payments for development milestones that take no account of the organization-wide effort to build valuable capital. In many organizations, managers with specific milestones to meet are understandably unwilling to permit the development of generalized capital assets for the company as a whole, if doing so puts them over budget on their particular projects. High-level management must recognize that developing working capital for software development is an investment in future productivity that deserves their support.

## 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>c</u> ustomer <u>r</u> evenue
CC	refers to	<u>c</u> ustomer <u>c</u> ost
N	refers to	<u>n</u> umber of customers in the market for that kind of software
WC	refers to	<u>w</u> orking <u>c</u> apital
DC	refers to	<u>d</u> evelopment <u>c</u> ost

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.

## Appendix

# Change Tolerance Through Modularity

*Note: The contents of this appendix are adapted directly from sections 3 and 4 of Chapter 4 of Software as Capital, by Howard Baetjer, Jr. (IEEE Computer Society Press, 1998).*

It is generally accepted in software engineering that modularity is crucial to the change tolerance of software, what I call “evolvability” in this appendix. Why? How does modularity facilitate software evolution? What aspects of modularity are important, and how are they related to characteristics of the social learning process? These are the questions we take up in this appendix.

---

### How modularity promotes evolvability

Simply stated, modularity leads to evolvability for two related reasons: 1) it reduces the amount of change necessary, and 2) it makes more understandable what must be changed. The first point seems to be well appreciated in software engineering circles, but the second is far more important. In order for a software system to evolve smoothly as its users and builders learn how it could be better, its overall structure must allow the system engineers to change it without too much difficulty. The enhancements can be addition of new capabilities, replacement of some part of the system's functionality with better, substantial redesign of the system, or some combination of these. When software architecture is appropriately modular, with functionality encapsulated in relatively independent modules, making the necessary changes is relatively easy, because they are confined to few modules, and therefore relatively easy to identify. In non-modular architectures, by contrast, there are lots of interdependencies among different parts of the system. These make the adaptation or extension very hard to accomplish, because so many different parts of the system are affected that it is not clear what must be done.

The *amount* of work involved is not the main issue; the issue is *understanding* what work must be done. True, where there are lots of interdependencies among different parts of the systems (we don't call them modules because the existence of many interdependencies implies that the system is *not* modular), there will be more work to do restoring the system's integrity when a change is made. But a far more important problem than the sheer volume of recoding to do is the danger that what recoding must be done will not be clear. A non-modular system will be significantly more difficult to understand than a modular one. Accordingly, when functionality is added or changed, or when restructuring is necessary, it is not clear what parts of the system are affected, and a great deal of effort must be expended finding out where problems remain.

Software development is a learning process; *if a system cannot be understood, then further learning in respect to it is hindered*. In extreme cases of multiple interdependencies in large systems, the system becomes literally incomprehensible; then adding or changing functionality in any but trivial ways is so difficult that the task is not one of change, but of beginning again and recreating the system entirely. That particular software species, so to speak, becomes extinct, because it can no longer adapt to the changing climate of business.

Modularity makes possible the evolution of extremely complex systems because modularity allows people to understand the system in pieces at various levels of abstraction. Each module is understandable as an entity on its own, and the overall system structure is understandable in terms of the relationships among these entities. While no one can understand a whole system in its entirety all at once, in order to maintain, extend, or renew the system it is necessary only to understand clearly defined pieces of the whole and their interrelationships with near neighbors.

An important factor here is the limitation on what participants in the development process need to know. This is called *information hiding*; we take it up in more detail below. Information hiding facilitates division of knowledge in the development process by making it unnecessary for a programmer working on one module to know very much about another module. Generally speaking, all one needs to know is what services a module provides, and how to ask for those services. *How* those services are provided is irrelevant.

Finally, appropriate modularity promotes evolvability because it leads to decentralized rather than hierarchical architectures, making it is easier to add functionality. Traditional design approaches frequently involve functional decomposition, in which a central function or purpose for the system is systematically decomposed into subprocesses at ever more fine-grained levels. In such architectures, it is difficult to add pieces without reconstructing much of the whole. Modular architectures, by contrast, tend to be designed by representing the various parts of the system being modeled. With such decentralized architectures, the pieces have a more equal relationship; the structure is more organic. Adding functionality is more like adding a node to a network than reconstituting a rigid skeleton.

---

## Kinds of modularity

What, exactly, do we mean by modularity? What are its aspects? There are several, and some are in tension with others. In fact, designers must often decide among different aspects of modularity when conflicts arise. The following list comes from Bertrand Meyer's well-regarded *Object-Oriented Software Construction*. These are Meyer's criteria for helping evaluate design methods with respect to the modularity they yield (1988, p. 12ff.).

### ***Modular decomposability***

This is the ability to decompose a problem into several subproblems, each of which may be worked on separately. This kind of modularity is essential to take advantage of specialization and the division of knowledge. If different individuals or teams are to be able to work on a problem at the same time, that problem must be decomposable into subproblems.

### ***Modular composability***

Quoting Meyer,

A method satisfies the criterion of Modular Composability if it favors the production of software elements which may be freely combined with each other to produce new systems, possibly in an environment quite different from the one in which they were initially developed (1988, p. 13).

Composability is a matter of what economists call "multiple specificity," as opposed to single specificity, the ability of a capital good to function in (be specific to) more than one context. If we are to take advantage of division of knowledge, then we need to depend on others' contributions, and we would like to enable sharing across time and place through embodiment of knowledge in composable modules. Where modules are composable, then it is not necessary to build anew when a new need arises for the functionality they provide. Composability provides economies of scope in design: one design can serve several purposes.

Note that composability may be at odds with decomposability: decomposing a problem into finer and finer subproblems may yield modules highly specific to the problem at hand, not generally applicable to other kinds of problems.

### ***Modular understandability***

This is the ability of a module to be understood on its own by a human reader, or with reference to at most one or two related modules. Code is not modularly understandable if it is meaningless except in context. It is modularly understandable if one can perceive what it does even in isolation from other modules. Understandability is a communication and coordination issue, important because software development is a social process. Whenever more than one person works on a software system, or even when a single person works on a system over time, coming back later to code that she wrote some time before, understandability is important, because it reduces the knowledge overhead for each individual who works on it. Consequently understandability is also a division of knowledge issue: if understanding different modules does not require knowledge of many others at the same time, it is easier for programmers to specialize on particular modules.

Understandability is of course essential during maintenance, when programmers other than those who built the code have to work on it. Generally, modules that correspond to identifiable abstractions in the real world tend to be more understandable than those that do not.

### ***Modular continuity***<sup>31</sup>

This is the characteristic that small changes in problem specifications require changes in only one or a few modules. It has fundamentally to do with localization of change. In everyday terms, a small change in specifications should require only a little bit of work. An illustrative counter-example of continuity is the great disturbance caused in many non-modular business software systems when the U. S. Post Office switched from five-digit zip codes to the present nine-digit zip code. Many software systems did not localize their treatment of zip codes, and had to be extensively rewritten at great expense.

Continuity is important because the learning process of software development does not stop. What the software must do will change; the more easily these new needs may be accommodated, the better.

### ***Modular protection***

Quoting Meyer again,

A method satisfies the Modular Protection criterion if it yields architectures in which the effect of an abnormal condition occurring at run-time in a module will remain confined to this module, or at least will propagate to a few neighboring modules only. (1988, p. 17)

Modular protection might at first seem insignificant to the software development process as such, because it concerns run-time problems rather than development problems. But there is an important implication for software development, given that software development is an uncertain, somewhat experimental process. That is, where there is modular protection and errors tend not to spread, programmers feel more free to experiment and hence to discover solutions. Ward Cunningham reports, for example, that in his team's development of the WyCash+ portfolio management package, which is built in Smalltalk with careful attention to modularity, they sometimes attempted major rearchitecting of the system. Sometimes the attempt would fail and they would have to revert to a previous version, but on other occasions they could accomplish very significant change with surprising ease.<sup>32</sup> By contrast, one frequently hears that programmers who work on large programs built with conventional techniques and without the support of object-oriented languages are "terrified to make changes because they are afraid that it will break."<sup>33</sup>

---

<sup>31</sup> Meyer takes the term by analogy to continuity of functions in mathematics, in which small changes in variables lead to small changes in results.

<sup>32</sup> Personal interview, October 1992.

<sup>33</sup> This observation was made to me by Bill Waldron of Krautkamer Branson in informal conversation. Krautkamer Branson builds ultrasonic flaw detection devices, using the C language for their software.

Software development is a learning process, during which the software developers learn a great deal from their interaction with their developing design. Frequently they learn that some prior design decision was flawed in such a way as to make the system less evolvable over time. If they are afraid to try to improve their designs at this stage, as they learn, fearing disastrous results, the software will come to embody less knowledge now, and less evolvability for the future. Modular protection thus fosters effective software evolution in an important way.

---

## Design principles that yield modularity

We have examined the benefits of modularity in software systems; let us turn to the practical matter of how modularity may be achieved. From a slightly broader perspective, this is a matter of asking what kinds of characteristics enable software to evolve well. Putting it metaphorically, we are asking what makes software flexible. In terms of lean development, we are asking what makes software change tolerant.

Kent Beck has observed that, "when you are in a brittle medium," it is important to do separate analysis and design on any software project before beginning coding, in order to avoid downstream costs and problems.<sup>34</sup> (Such prior analysis and design is often necessary despite the problem that necessary knowledge is often unavailable until users have a chance to see and use a running version. The reason is that when program development is done in an unforgiving programming language, there may be no alternative.) One of the main downstream costs of working in a brittle medium is the plain inability to make changes one would wish to make. One designer at IBM observed that C programs often stay unwieldy and difficult to work with, because even when a team perceives some kind of major change they wish to make to clean up the design and make it more understandable and simple, they must nevertheless proceed with their current, inferior design, because to get the program the way they would like it would require too much change.<sup>35</sup> (This designer was working, at the time he made the comment, on a system built in C. He wished to return to Smalltalk, with which he claimed he could be *ten times* as productive).

When one is in a flexible medium, however, such as a good object-oriented environment and design, it becomes far more possible to let analysis, design, and implementation occur together, without encountering excessive downstream costs and problems. Keeping analysis, design, and implementation together in an iterative development method is extremely valuable because of the nature of knowledge and learning: Because so much knowledge is tacit, it is problematical to try to analyze all first. Because developers learn by doing, it is problematical to try to finish designing before implementing. When the medium is flexible enough, it is not so costly to make changes downstream as one learns. In brief, maintenance is easier.

---

<sup>34</sup> Personal interview, October 1992.

<sup>35</sup> Lee Griffin of IBM Corp., personal conversation.

What are the design characteristics that allow software to evolve, that allow new knowledge to be built in smoothly? Again following Meyer, we can identify five, and we quote his statement of the modularity principles in each case (1988, pp. 18-23). Each of these principles is rooted in the social nature of software development: For software to be extended and enhanced, people must understand it and work on it, generally in groups. These principles facilitate that group effort.

### ***Linguistic modular units***

"Modules must correspond to syntactic units in the language used."

This principle requires direct mapping of terms in the programming language to design elements (and further, ideally, to real world entities being modeled in the software system). Sometimes this feature is known as "proximity to the problem space": the terms used in the program refer directly to modules of the system, which represent elements in the problem space. In business programs, for example, there might be modules such as `PurchaseOrder`, `Customer`, and `CreditCardCompany`. In a design that holds to the principle of linguistic modular units, real world purchase orders would be represented by separate purchase order modules in the software, in which `PurchaseOrder` is a distinct syntactic unit.

The crucial benefit of linguistic modular units is that *they make it easier to think about and understand the elements of software systems*. This is important both in helping individual programmers understand the systems they are working on, and in enriching the dialogue among designers, users, and programmers, who can use the same terminology in describing the system from their different points of view. Kent Beck has recently written that "[t]he cost of a piece of code over its many-year life is dominated by how well it communicates to others. If it is easy to understand, it will cost your company less while bringing the same benefits." (Beck, 1995, p. 20)

To see the value of this principle, consider that in older programming languages, modules frequently were not identified linguistically within the programming language. They might stretch, say, from line 450 to line 755, and be accessed by a statement such as, "`GOTO line 450.`" The necessity of memorizing what happens in the module obstructs programmers' progress; it is much easier to work with a statement such as, "`PurchaseOrder new initialize.`"

In respect to communication between technical and non-technical people, linguistic modular units make it easier for non-programmers to participate in the dialogue. If the creation of a purchase order is handled by lines 450 to 755, the programmer is likely to think *in terms of* that chunk of code rather than the concept "purchase order." The client, however, will know nothing of lines 450 to 755 as such; he will think instead of actual purchase orders. Hence they have communication problems. On the other hand, when they both can speak of "purchase order," the one thinking of the module in the code, and the other thinking of the physical item represented by it, communication is greatly facilitated.

### ***Few interfaces***

"Every module should communicate with as few others as possible."

The more interconnections there are between modules, the more likely it is, when one of them needs to be changed, that others will have to be changed also. When many modules have to be changed, there is more likelihood that the people who have to change them will lose track of all that needs to be done. This means expense, delay, and greater likelihood of errors. Thus, for the sake of continuity, the number of interfaces should be restricted. *Restricting the number of interfaces helps maintain the division of knowledge*, because those responsible for interacting modules must coordinate with one another when changes are made, and if only a few modules interact, there is less coordinating to do, less propagation of change.

Having numerous interfaces with their associated rigidities is a common consequence of centralized designs. Generally speaking, in centralized, top-down structures, most of the modules at the periphery need to communicate in some fashion with the modules at the core, which are responsible for reconciling the interactions of all. The soviet-type economy comes to mind. The difficulty is that everything depends on proper operation at the center, and if a problem occurs there or some change becomes necessary, everyone is affected. Furthermore, centralized structures imply some fundamental, overarching purpose.

By contrast, there are

...more "libertarian" structures, [in which] every module just "talks to" its two immediate neighbors, but there is no central authority. Such a style of design is a little surprising at first since it does not conform to the traditional model of functional, top-down design. But it may be used to obtain interesting, robust architectures; this is the kind of structure that object-oriented techniques tend to yield (Meyer 1988, p. 47).

In such structures, dependencies are greatly reduced. Additionally, these structures lend themselves to systems in which there is not one clear purpose, but rather a variety of different services that the software may provide its users. As an economy has no central purpose, and therefore functions best according to decentralized interactions among the agents that constitute it, so also many software systems have no central purpose, and therefore are best structured in a decentralized manner. Good examples of such systems are the increasingly popular "enterprise models." These are essentially software representations of an entire enterprise. The modules represent, say, different divisions of a business or different processes that occur within them, and the interfaces among modules represent the interactions among related parts of the business.

### ***Small interfaces (weak coupling)***

"If any two modules communicate at all, they should exchange as little information as possible."

Meyer's statement of this principle is overstated. The point of this principle, as of the last, is to reduce dependencies, rather than to reduce communication. The difficulty this principle seeks to avoid is having modules depend on a large amount of shared information -- more than what they actually need to interact usefully. *Small interfaces also foster the division of knowledge.*

There are a number of difficulties with extensive dependencies. One is that modules become "tightly coupled" in depending on a lot of the detail of one another, or on some shared data source. Such tight couplings diminish the division of knowledge and hence evolvability, because when some part of that detail or data changes, all affected modules must be rewritten, and when errors occur, they propagate widely. Furthermore, when one module has access to too much of the detail of another module, there is the danger of interference. What this means in practice is that in the development process, programmers will be tempted to use too much of the available, detailed information about other modules in the design of their own, or even use it inadvertently, without even being aware that they are doing so. The danger is no less for experienced programmers than for inexperienced, because the experts might be additionally tempted to "make clever use" of some of that information, which may later change. Simply put, this principle holds that modules should be as independent as possible.

Object-oriented techniques address this issue by the equivalent of property rights to data (Miller and Drexler 1988), achieved through encapsulation of data and message passing. No object may directly access some other object's data; that is private and contained within the object. Instead, one object gains the services of another through passing a message: the message contains only the data needed by the service-providing object, and the response contains only the data specifically asked for by the client.

Objects communicate what they have and what they can offer; what they pointedly do not communicate are any details of how they work. For this reason they are known as "abstract data types."

Using abstract data type descriptions, we do not care (we refuse to care) about what a data structure *is*; what matters is what it *has* -- what it can offer to other software elements. ...[T]o preserve each module's integrity in an environment of constant change, every system component must mind its own business (Meyer 1988, p. 54).

Restricting the amount of information that passes across an interface is an aspect of *information hiding*, an important element of modular programming, which we take up in more detail below.

### ***Explicit interfaces***

"Whenever two modules A and B communicate, this must be obvious from the text of A or B or both."

The reason for this principle is clear: For people to work with modules effectively, it must be clear what each module does and where interdependencies among them lie. Few problems so hinder smooth evolution of a system as hidden interactions that cause unexpected effects. Ideally, the communication between modules should be obvious from the text of both. Explicit interfaces *help developers understand the relationships among system modules.*

### **Information hiding**

"All information about a module should be private to the module unless it is specifically declared public."

Information hiding dramatically reduces the complexity that programmers face and the cognitive demands on them. In a manner suggested by our discussion of small interfaces above, it allows programmers to ignore the contents and functioning of modules they call on. Programmers are thereby freed to think simply about what services those modules provide. Hence *information hiding allows software developers to understand the system better by viewing it at a high level of abstraction*, not getting confused in the trivial detail. While information hiding decreases the likelihood that a programmer might improperly try to change another module,

[T]he purpose of information hiding is abstraction, not protection. We do not necessarily wish to prevent client programmers from accessing secret class elements, but rather to *relieve* them from having to do so. In a software project, programmers are faced with too much information, and need abstraction facilities to concentrate on the essentials. Information hiding makes this possible by separating function from implementation, and should be viewed by client programmers as help rather than hindrance (Meyer 1988, p. 204).

Separation of interface and implementation is the essence of information hiding. The interface -- the messages or routines through which a module interacts with others -- must of course be publicly known. But its implementation, the methods it uses to carry out its tasks and the data structures it draws on, should be private. Others should not need to know them. An important benefit is that when a module's implementation changes, other modules are not affected. As long as the object in question responds to the same message, other objects calling on it for services are not affected.

In object-oriented languages, one way in which information hiding is accomplished is through the combination of polymorphism and dynamic binding. Wide varieties of related objects may be called on polymorphically, i.e., with the same interface, which captures some abstraction they share. For example, doors, windows, books, and mouths may all be *shut*. The same term *shut* applies polymorphically (in a variety of forms) to each. Of course there is a different procedure for each variety of *shut*, corresponding to the different (kinds of) objects, but that procedure may remain hidden from those who write the client code. The right procedure is applied to each through dynamic binding.

The great benefit that polymorphism and dynamic binding provide programmers and programmer teams trying to evolve software is that the combination allows them to concentrate on the essential abstractions and not get lost in the detail of implementation. They can use their natural faculties for conceptualization and abstraction and apply them directly to the problem they are working on, comfortably removed from the nitty-gritty requirements of the computers.

An illustration of the benefit comes from the recent experience of Texas Instruments in building a new computer-integrated-manufacturing system for manufacture of semiconductors. They built the system to control fabrication machines built by Texas Instruments, but at a late point in system development they had to extend the system to control fabrication machines built by a third-party supplier also. It was not necessary to build a separate system to control the different machines. They used the same interface for the third-party machines as they used for the TI machines; all they had to tailor to the needs of the third-party machines was the new implementation code.<sup>36</sup>

---

<sup>36</sup> Experience report presented at OOPSLA 1992 by John McGehee of Texas Instruments.

## Bibliography

- Adams, Sam S. 1992a. "Software assets and the CRC technique," *Hotline on Object-Oriented Technology*, Vol. 3, no. 10, August.
- Adams, Sam S. 1992b. "Object-oriented ROI: extending CRC across the lifecycle," *Hotline on Object-Oriented Technology*, Vol. 3, no. 11, September.
- Adams, Sam S. 1992c. "Software Reuse and the Enterprise," *Software Development '92*. Spring Proceedings.
- Allen, Peter M., 1990. "Why the Future Is Not What It Was," prepared for *Futures*, 6/4/90. Bedford, England: International Ecotechnology Research Center.
- Barn, Balbir S. 1992. "User Interface Development: Our Experience with HP Interface Architect," in Spurr, Kathy, and Layzell, Paul, eds., *CASE, Current Practice, Future Prospects*, Chichester: Wiley.
- Beck, Kent. 1995. "Clean code: Pipe dream or state of mind?" *The Smalltalk Report*, Vol. 4, no. 8, June.
- Harris, Kim. 1991. "Hewlett-Packard Corporate Reuse Program," *Proceedings of the Fourth Annual Workshop on Software Reuse*, Reston, Virginia, Nov. 18-22, 1991.
- Hayek, F.A. 1935. "The Maintenance of Capital," in *Profits, Interest and Investment*, London: Routledge & Sons.
- Hayek, F.A. 1945. "The Use of Knowledge in Society," in Hayek (1948).
- Hayek, F.A. 1979. *The Counter-Revolution of Science*, Indianapolis: LibertyPress.
- Kara, Daniel A. 1992. "CASE and Advanced Software Development on the Macintosh," *CASE Trends* Vol. 4, no. 6, September.
- Knight, Frank H. 1971 [1921]. *Risk, Uncertainty, and Profit*. Chicago: University of Chicago Press.
- Lachmann, L. M. 1975. "Reflections on Hayekian Capital Theory," Paper delivered at the Allied Social Science Association meeting in Dallas, Texas. Photocopy 1975.
- Lachmann, L. M. 1986. *The Market as an Economic Process*. New York: Basil Blackwell.
- Menger, Carl. 1981 [1871]. *Principles of Economics*. New York: New York University Press.
- Meyer, Bertrand. 1988. *Object-oriented Software Construction*, Englewood Cliffs, NJ: Prentice-Hall.

- Meyer, Bertrand. 1990. "The New Culture of Software Development" *Journal of Object-Oriented Programming* (Nov./Dec.)
- Meyer, Bertrand. 1991. "From the bubbles to the objects," in "Evolution vs revolution: Should structured methods be objectified?" *Object Magazine*, 1:4, November/December.
- Miller, Mark S. and Drexler, K. Eric. 1988. "Markets and Computation: Agoric Open Systems," in B.A Huberman, ed., *The Ecology of Computation* Amsterdam: North-Holland.
- Mullin, Mark. 1990. *Rapid Prototyping for Object-Oriented Systems*, Menlo Park, California: Addison-Welsey.
- Polanyi, Michael. 1958. *Personal Knowledge*, Chicago: University of Chicago Press.
- Polanyi, Michael. 1964. *The Tacit Dimension*, New York: Doubleday.
- Robinson, Keith. 1992. "Putting the SE into CASE," in Spurr, Kathy, and Layzell, Paul, eds., *CASE, Current Practice, Future Prospects*, Chichester: Wiley.
- Ryan, Doris. 1991. "RAPID/NM," presentation at the *Fourth Annual Workshop on Software Reuse*, Reston, Virginia, Nov. 18-22, 1991.
- Salin, Phil. 1990. "The Ecology of Decisions, or, 'An Inquiry into the Nature and Causes of the Wealth of Kitchens,'" *Market Process*, 8: 91-114.
- Smith, Adam. 1976 [1776]. *An Inquiry Into the Nature and Causes of the Wealth of Nations*. Chicago: University of Chicago Press.
- Smith, M.F. 1991. *Software Prototyping: Adoption, Practice, and Management*, London: McGraw-Hill.
- Sowell, Thomas. 1980. *Knowledge and Decisions*. New York: Basic Books.
- Taylor, David A. 1990. *Object-Oriented Technology: A Manager's Guide*, Alameda, California: Servio Cororation
- Tirso, Jesus. 1991. "IBM Reuse Program," *Proceedings of the Fourth Annual Workshop on Software Reuse*, Reston, Virginia, Nov. 18-22, 1991.
- Wheelwright, Steven C., and Clark, Kim B. 1992. *Revolutionizing Product Development*. New York: The Free Press.
- Whitefield, Bob and Auer, Ken. 1991. "You can't do that in Smalltalk! Or can you?" *Object Magazine*, Vol. 1, no.1, May/June.
- Womack, James P., Jones, Daniel T., and Roos, Daniel. 1990. *The Machine that Changed the World*, New York: Harper Perennial.