

## INFORMATION TO USERS

This manuscript has been reproduced from the microfilm master. UMI films the text directly from the original or copy submitted. Thus, some thesis and dissertation copies are in typewriter face, while others may be from any type of computer printer.

**The quality of this reproduction is dependent upon the quality of the copy submitted.** Broken or indistinct print, colored or poor quality illustrations and photographs, print bleedthrough, substandard margins, and improper alignment can adversely affect reproduction.

In the unlikely event that the author did not send UMI a complete manuscript and there are missing pages, these will be noted. Also, if unauthorized copyright material had to be removed, a note will indicate the deletion.

Oversize materials (e.g., maps, drawings, charts) are reproduced by sectioning the original, beginning at the upper left-hand corner and continuing from left to right in equal sections with small overlaps. Each original is also photographed in one exposure and is included in reduced form at the back of the book.

Photographs included in the original manuscript have been reproduced xerographically in this copy. Higher quality 6" x 9" black and white photographic prints are available for any photographs or illustrations appearing in this copy for an additional charge. Contact UMI directly to order.

# U·M·I

University Microfilms International  
A Bell & Howell Information Company  
300 North Zeeb Road, Ann Arbor, MI 48106-1346 USA  
313/761-4700 800/521-0600



**Order Number 9316545**

**Software as capital: Lessons for economic development from  
software engineering**

**Baetjer, Howard, Jr., Ph.D.**

**George Mason University, 1993**

**U·M·I**

300 N. Zeeb Rd.  
Ann Arbor, MI 48106



**SOFTWARE AS CAPITAL:  
LESSONS FOR ECONOMIC DEVELOPMENT  
FROM SOFTWARE ENGINEERING**

A dissertation submitted in partial fulfillment of the requirements for the degree of  
Doctor of Philosophy at George Mason University.

by

**Howard Baetjer Jr.  
B.A., Princeton University 1974  
M.Litt., University of Edinburgh 1980  
M.A., Boston College 1984**

**Director: Professor Don Lavoie  
Program on Social and Organizational Learning**


**Spring 1993  
George Mason University  
Fairfax, Virginia**

SOFTWARE AS CAPITAL:  
LESSONS FOR ECONOMIC DEVELOPMENT  
FROM SOFTWARE ENGINEERING

by

Howard Baetjer Jr.  
A Dissertation  
Submitted to the  
Faculty of the Graduate School  
of  
George Mason University  
in Partial Fulfillment of  
the Requirements for the Degree  
of  
Doctor of Philosophy  
Economics

Committee:

  
\_\_\_\_\_  
Jack High  
\_\_\_\_\_  
Brad Cox  
\_\_\_\_\_  
Karen J. Vaughn  
\_\_\_\_\_  
David C. Bine  
\_\_\_\_\_  
Richard E. Wagner  
\_\_\_\_\_  
Kingsley L. Haynes

Director

Department Chairman

Dean of the Graduate School

Date:

1/11/93

Spring 1993  
George Mason University  
Fairfax, Virginia

for  
Johnnie, Markie, Asia, and Jessica,  
who brightened the process

## Acknowledgments

I am profoundly grateful to the Center for the Study of Market Processes for five years of financial, intellectual, and emotional support of my academic efforts, which have made this paper possible. Thanks ever so much. I am grateful as well to the Claude R. Lambe Foundation and the J.M. Foundation for financial support.

Kent Beck and Ward Cunningham each provided an extremely valuable interview: many thanks to them both. Thanks to Tom Wrensch for helping me learn Smalltalk and understand better the software development process. I am also indebted to Paul Ambrose, Lee Griffin of IBM, Richard Collum of First Union National Bank of North Carolina, and Bill Waldron of Krautkamer Branson for short, but valuable conversations about the software development process. Thanks to Robert Polutchko of Martin Marietta Corp. for a short conversation which provided useful insight into the nature of software.

Thanks to Phil Salin, founder of the American Information Exchange Corp., and to all my teammates at AMiX, both for the chance to share in bootstrapping an electronic market for software components and for providing an electronic marketplace in which I was able to purchase research assistance. I am grateful to Dr. Brad Cox for his steady support of my work, well before he joined my committee, and to Professors Karen Vaughn, Jack High, and David Rine for their helpful comments and criticisms.



Thanks to Mark Miller and Eric Drexler for opening my eyes to this general area of research, and to Mark Miller in particular for extensive interviews, inspiration, and support.

Thanks to the designers of Microsoft Word for Windows, for providing me with a superb tool for dissertation-writing.

Thanks to Kevin Lacobie for research help and for numerous valuable discussions.

I am extremely grateful to Bill Tulloh for a host of valuable suggestions and insights into this topic, for inestimable help with research, for criticism of earlier drafts, and for his enthusiasm for the topic.

My primary debt of gratitude is to my chairman, Don Lavoie. For his steady support and encouragement, for intellectual inspiration, for his great generosity in time, attention, and timely response to every request for guidance, and in particular for the example of his passion for understanding, I am extremely grateful.

## Table of Contents

	Page
Abstract .....	vii
Preface .....	1
Chapter 1. Knowledge Capital and the Theory of Economic Growth.....	4
1. The subject of this investigation: Better tools as a cause of the wealth of nations .....	4
2. Irrelevance of mainstream "theory of economic growth" .....	6
3. Capital goods as knowledge.....	24
4. Capital goods and division of knowledge across time and space ...	40
5. Capital structure.....	45
6. Capital development as a social learning process.....	51
Chapter 2. A Short History of Software Development .....	54
1. Introduction .....	54
2. Overview.....	56
3. The Key Challenge: Managing Complexity .....	58
4. The Evolution of Programming Practice .....	60
5. Object-Oriented Technologies .....	70
6. Summary .....	76
Chapter 3. Designing new capital: lessons from software development.....	78
1. Introduction .....	78
2. Discovery in the design process: why prototyping .....	79
3. Designing as understanding: the role of tools for thought.....	98
4. Summary .....	118
Chapter 4. Capital Evolvability: Lessons from Software Maintenance .....	120
1. Introduction .....	120
2. Evolvability as a design goal.....	124
3. Evolvability through modularity .....	132
4. Design principles that yield modularity .....	138
5. Accelerating evolution through software reuse .....	147

Chapter 5. Evolving the Capital Structure: Markets for Software Components .....	155
1. Introduction .....	155
2. Component markets as an unintended consequence of improved modularity .....	158
3. Learning through markets.....	160
4. Aspects of component market evolution .....	169
5. Summary .....	179
 Chapter 6. Conclusions: Implications for Economic Development and for Growth	
Theory.....	182
1. Introduction .....	182
2. Applicability to hard tools .....	182
3. Implications for economic development: exponential growth?....	189
4. Implications for growth theory .....	193
 Bibliography .....	200

## Abstract

### SOFTWARE AS CAPITAL: LESSONS FOR ECONOMIC DEVELOPMENT FROM SOFTWARE ENGINEERING

Howard Baetjer Jr., Ph.D.

George Mason University, 1993

Dissertation Director: Professor Don Lavoie

This dissertation investigates the nature of economic development: how do human societies advance in economic well-being? More narrowly, it investigates the role of capital goods in this advance: what is the nature of the processes by which people improve the capital structure? The dissertation addresses these questions through examining software development – its practices, tools, and technologies, and their evolution. Software development illuminates the nature of capital development in general.

The theoretical foundation of the work is Austrian capital theory. Neoclassical growth theory, including the "new growth theory" of Paul M. Romer, is found to be of little use here, because it abstracts away from what Ludwig Lachmann calls the structural aspects of capital: its heterogeneity and complementarity. Following Carl Menger, this study attributes human advancement primarily to the advancement of human knowledge. Capital goods embody the knowledge of many. Because this

knowledge is dispersed, often tacit, and incomplete, the capital structure evolves through a social learning process.

Requirements for software applications can rarely be established before development begins. Hence software developers have evolved procedures such as rapid prototyping for learning what the software must do. Prototyping is an iterative, interactive, dialogue-like learning process. Tools, languages, and methodologies are evolving which primarily enable understanding of the complex systems being developed. Object-oriented programming systems especially improve learning by providing rapid feedback and by allowing developers to program with high-level abstractions suitable for thinking about the problems being addressed.

Because conditions change, software must constantly evolve to maintain its value. Experience with software maintenance suggests that evolvability of software systems is fostered by modularity, which improved understandability, localized changes, and reduces system complexity. Modular, object-oriented techniques also enable construction of reusable software components, and hence improved specialization and division of knowledge.

Availability of reusable components may lead to component markets, given requisite social learning in the development of standards, new distribution channels and pricing techniques. Markets themselves would foster additional social learning by generating new information.

Exponential growth is checked by the difficulty of social learning: more rapid economic development depends on learning better at all levels of economic organization.

## Preface

The fundamental aim of this dissertation is to achieve a better understanding of economic development. More particularly, the aim is to understand the processes through which the capital structure – our tools of production and their interrelationships – develops and improves. The dissertation seeks, accordingly, to contribute both to the theory of economic development and to capital theory, (indeed, I view these as inextricably related), by addressing the problem of how new and better systems of production are conceived and built. Why, then, does the paper focus primarily on software development?

The reason is that knowledge is prominent in software as in no other kind of capital good. Following Carl Menger, I view capital goods as being fundamentally embodied knowledge. This dissertation will emphasize this view, and strive to explicate the processes by which knowledge is elicited, discovered, and embodied in the capital structure. With most other kinds of capital goods it is easy to overlook how much knowledge is built into them, because what we see is the steel and glass, the copper and plastic, the silicon and polymer in which that knowledge is embedded. Software, by contrast, we do not see at all; we think about it independent of its physical form. We are equally comfortable thinking about it as printed out on paper, stored magnetically on a floppy disk, or loaded and running in the circuits of a computer. Indifferent to the physical medium in which it is embodied, we are readily able to focus on the knowledge that software embodies.

While the paper focuses on software, the principles uncovered apply to capital goods in general. All kinds of capital goods are embodied knowledge; for all of them knowledge is of the essence, not the physical substrate on which that knowledge is imprinted. The software development experience has parallels in the development of physical capital. In the concluding chapter, we will take up these parallels directly.

The first chapter lays the theoretical foundation for the investigation. After pointing out that neoclassical growth theory offers little insight into the role of capital in economic development, it draws on Austrian capital theory for understanding of the relationships between knowledge and capital, and of the structural aspects of capital. It finds that, given the nature of capital goods, capital development is a social learning process. Chapter 2 then provides an overview of the history and terminology of software development, introducing some of the main issues which subsequent chapters will investigate for illumination of this process.

Chapter 3 discusses initial software development, looking particularly at the evolution of the procedures and tools used. It demonstrates that the software development process is one of interactive, social learning. Chapter 4 goes on to investigate the on-going development of software – software maintenance – that occurs after initial products are delivered. The purpose of the chapter is to assess the characteristics of software which is able to evolve readily; we find modularity to be of fundamental importance.

Chapter 5 broadens the perspective to examine the prospects for software component markets and the benefits those markets promise. It treats the evolution of such markets, and the institutions and attitudes requisite thereto, as another



species of social learning. Chapter 6 summarizes, draws parallels to the development of physical capital goods, and draws implications about the rate economic development, both for growth theorists who would study it, and for practitioners who would improve it.

## Chapter 1

### Knowledge Capital and the Theory of Economic Growth

*Men, my brothers, men, the workers, ever reaping something new,  
That which they have done but earnest of the things that they shall  
do.*

*For I dipped into the future, far as human eye could see,  
Saw the Vision of the world, and all the wonder that would be.  
- Tennyson, "Locksley Hall"*

#### **1. The subject of this investigation: Better tools as a cause of the wealth of nations**

The dissertation is motivated by the same question which motivated Adam Smith's *An Inquiry into the Nature and Causes of the Wealth of Nations* (1776): how do we account for human beings' economic advancement? How is it that our race of talking primates has been able to advance from barbarism to abundance (at least in certain areas of the world)? What is the nature of the process by which we are able, over time, to get more and better of the "necessaries and conveniences of life" for the same amount of effort?

My piece of this large inquiry takes as its point of departure the observation that human society advances in economic well-being by increasing its productivity per person and by extending trade, and that these improvements depend on appropriate rules of conduct. Human advancement is thus an intertwined evolution of the capital structure, the catallaxy, and the common law. I focus on the first of these

and ask, how do people in a society improve their productivity – their ability to produce more of the things they want with a given amount of human effort? They do so fundamentally by increasing their knowledge of productive relationships, and building this knowledge into better tools – better devices which extend their physical, perceptual, and mental faculties for understanding and transforming the world they live in. This view of human advancement I derive from the Austrian School of economics, and in particular from the founder of the Austrian School, Carl Menger.<sup>1</sup>

Of course we may improve productivity by working harder, but the effects of greater exertion are far less than the effects of better tools. In a task such as reaping grain, for example, even the most heroically increased exertions of a barehanded reaper yield far smaller productivity gains than equipping an average worker with a steel sickle. Also we may improve productivity by producing greater numbers of the same kinds of tools, but here again the effects fall far short of the effects of building better tools. Even if we were to equip everyone in the village with a steel sickle, productivity at harvest time must fall far short of what it would be if we were to equip only one worker with a John Deere grain combine. Better tools, then, are the key to greater productivity. For a society to improve its productivity, that society must improve the quality of its tools – its capital goods.

The context in which capital is meaningful is production. Production is a matter of transforming our condition from a less-preferred to more-preferred state. What

---

<sup>1</sup> Essential works in this tradition are Menger (1981), Bohm-Bawerk (1959), Hayek (1935 and 1941), Mises (1966), and Lachmann (1978).

transformations will answer the purpose, and how to carry them out, are the crucial questions. Any capital is going to be some kind of embodied knowledge of such transformations and how to accomplish them. Capital is saved-up learning which gives us a head start on production.

How does a society improve the quality of its capital goods? How does it manage to save up its knowledge of useful transformations? What is the nature of the process, and what is involved in the process? These are the questions to be explored in this dissertation.

## **2. Irrelevance of mainstream "theory of economic growth"**

Because tools are so important to economic development and growth, one might expect to find insight into these questions in the branch of economics known as the theory of economic growth. But in fact, with the exception of Joseph Schumpeter's work (1934), growth theory, both the traditional and the "new growth theory," is engaged in a different kind of inquiry. Growth theory has very little to say about the development of the new and better tools we ultimately depend on for economic advancement – the development of the capital structure. Notwithstanding the merits this body of work may have for understanding other aspects of economic growth, it has little relevance for the present inquiry.

### 2.1. Problematic aspects of traditional growth theory: the Harrod-Domar-Solow approach

At the center of neoclassical growth theory is the Harrod-Domar approach (Harrod 1939, Domar 1946 and 1957), which was elaborated by Robert Solow in work that

helped win him the Nobel prize. (1956, 1970) Although this body of work refers to capital extensively, it says very little about capital, and nothing about how the capital structure evolves. In fact, it assumes that the capital structure does not evolve in any qualitative way. There are three closely interrelated assumptions in this theory which necessarily eliminate consideration of actual improvements to capital goods and the capital structure.

### **It ignores the heterogeneity of capital**

A fundamental problem with the Harrod-Domar-Solow strand of growth theory for our purposes is that it treats capital as homogeneous. In Harrod's model, capital is a homogeneous stuff that can be accumulated incrementally. The "actual saving in a period . . . is equal to the addition to the capital stock,"<sup>2</sup> Harrod tells us. This indicates that quantities of "capital" may be indefinitely built up. Solow's discussion of the model makes this more explicit: he defines "the stock of capital" as "the sum of past net investments" (1970, p. 4, emphasis added), and says that the "capital requirement per unit of output [is a] fixed number . . . in the sense that [it does] not change in the course of time" (1970, p. 9). Capital is not only homogeneous in time, according to Solow, but also homogeneous across time.

This mechanical approach to capital treats it like a multiplier: more capital means a bigger number multiplying the effort of labor. E.g., if we have 100 units of K at time 0, and, say, 5 laborers, then we get  $5 * 100 = 500$  units of output. Then we take some savings from that output and (less depreciation) add it to the 100. Suppose

---

<sup>2</sup> Harrod (1939, p. 18). All references to Harrod are from this work.

net savings are 3, then in period 1 we have  $5 * 103 = 515$  units of output. Capital is essentially all of the same kind and quality. Its value is its purchase price; it can be increased only quantitatively. Given fixed input of human effort, getting more output with the same "amount" of capital is not possible.

But capital in the world is not homogeneous. As Ludwig Lachmann points out, "capital resources are heterogeneous. . . . While we may add head to head . . . and acre to acre . . . we cannot add beer barrels to blast furnaces nor trucks to yards of telephone wire." (Lachmann 1978, p. 2) Furthermore,

for most purposes capital goods have to be used jointly. Complementarity is of the essence of capital use. But the heterogeneous capital resources do not lend themselves to combination in any arbitrary fashion (1978, p. 3)

Some capital combinations are useless: beer barrels and blast furnaces, for example. But other combinations multiply the value of one another, e. g., fertile fields and advanced farm machinery. To quote from Lachmann again,

The theory of capital must therefore concern itself with the way in which entrepreneurs form combinations of heterogeneous capital resources in their plans, and the way in which they regroup them when they revise these plans. A theory which ignores such regrouping ignores a highly significant aspect of reality: the changing pattern of resource use which the divergence of results actually experienced from what they had been expected to be, imposes on entrepreneurs. (1978, p. 35)

The theory of capital must also concern itself with the way in which entrepreneurs develop new, different, and better heterogeneous capital resources.

We note in passing that the Harrod-Domar-Solow theory not only fails to differentiate between kinds and orders of capital, it also fails to differentiate even between capital goods and consumption goods. Harrod states that, "No distinction

is drawn in this theory between capital goods and consumption goods. In measuring the increment of capital, the two are taken together; the increment consists of total production less total consumption." (p. 18) This is another way of saying that saving equals investment, and that all savings automatically become capital goods. This failure to distinguish between the different categories of goods produced leads Harrod to such remarkable statements as, "a condition of general over-production is the consequence of producers in sum producing too little." (p. 24) In Solow's development of Harrod's work the blurring of capital goods and consumption goods is made even more explicit: "The model economy produces only one composite commodity, which it can either consume currently or accumulate as a stock of capital."<sup>3</sup>

Modeling production in this way helps illuminate the role of savings in economic development, and draws attention to interesting issues concerning the sustainability of growth rates under particular conditions. In particular it helps clarify the conditions under which a dynamic equilibrium might be possible. But the present investigation is concerned with how we develop new and better means of producing the things we want; therefore a theory that assumes away differences between the things we want and the means of producing them is not of use here.

#### **It assumes quantifiability of capital**

Traditional growth theory also relies on a mathematical treatment of capital: capital appears in the models as a numerical variable in a production function. Such a

---

<sup>3</sup> Solow (1970, p. 9). Unless otherwise noted, all references to Solow are from this work.

treatment implies that capital can be meaningfully quantified – measured in some way. Harrod, for example, speaks of "the value of ... capital goods" (p. 16) and posits that "actual saving in a period ... is equal to the addition to the capital stock" (p. 18). The terminology gives the impression that capital can be easily measured, and the equations depend on the economy's capital stock being quantifiable.

But capital is ultimately unmeasurable.<sup>4</sup> As Harrod's colleague Joan Robinson observed, "no one ever makes it clear how capital is to be measured."<sup>5</sup> Israel Kirzner addresses the immeasurability of capital in his An Essay on Capital (1966).<sup>6</sup> First he dispenses with the idea that capital can be measured in raw physical terms.

The truth is that the heterogeneity of the various physical items in the stock not only constitutes a well recognized barrier to the construction of such a measure, but represents at the same time the reason why such a measure can play no significant role at all in the analysis of decision making in the course of capitalistic production. The producer simply cannot afford to ignore the heterogeneity of the various items in the capital stock. (p. 105)

Kirzner then turns to "backward-looking" measures of the existing capital stock: the past sacrifices – the costs – involved in building up that stock. This is the kind of measure most in accordance with the Harrod-Domar-Solow methodology, since

---

<sup>4</sup> An economy's aggregate capital stock cannot be measured, although a firm's can be, in a sense: a firm can calculate the money value of its capital. Economic calculation achieves a rough and ready way of measuring the value of capital for a given profit center.

<sup>5</sup> Joan Robinson, The Rate of interest, and other Essays, p. 54, quoted in Lachmann (1978), p. 5.

<sup>6</sup> See also Hayek (1935) and Lachmann (1975 and 1986).



they declare the value of new capital to be that of the output not consumed in a period. Kirzner points out that these past costs are generally of different kinds and made at different dates; accordingly they cannot be meaningfully summed.

Likewise "forward-looking" measures of capital are unsatisfactory. These are the efforts "to measure the capital stock by the contribution to future production that it is able to make" (p. 113). With these measures there are a number of difficulties, the most important being that future value depends on many individuals' plans for the capital (which has alternative uses), and that these plans may be mutually inconsistent. "[I]t is in many respects a misleading simplification to talk as if a given resource were unambiguously associated with a definite flow of output, in the sense that such an output flow is forthcoming automatically from the resource." (p. 114).

The point here is not simply that it is technically difficult to quantify the amount or the value of capital, but that the notion of an amount of capital has at best an extremely imprecise meaning. It is imprecise even as an accounting measure within a firm, where plans for the use of different pieces of capital can be kept more or less compatible. But as the level of aggregation increases, the imprecision grows rapidly. "The amount of capital" is at best a useful mental shorthand. Treating it as if it were precise, in a mathematical equation, is more likely to confuse than to clarify.

**It assumes a fixed functional relationship between aggregate capital and output**

The two problems mentioned above – the twin assumptions of homogeneity and of quantifiability of capital – are probably consequences of this third: that the theorists are determined to represent the relationship between capital and output as a

functional relationship, in which the function itself is not allowed to change.<sup>7</sup> To model production in a functional relationship with capital necessitates an interpretation of capital as homogeneous, so that it may be aggregated meaningfully, and as quantifiable (at least in principle), so that this aggregation may be represented by a numerical variable.

The only place Harrod's model<sup>8</sup> might admit changes in the quality of people's tools, and hence in the production function, is in  $C$ , the "capital coefficient," defined as "the value of the capital goods required for the production of a unit increment in output." But as Harrod presents  $C$ , changes in its value are

---

<sup>7</sup> Technological change may sometimes occur, but only as an exogenous shock.

<sup>8</sup> Harrod's 1939 article, "An Essay in Dynamic Theory" entails the following elements:

- $G$  the geometric rate of growth of output or income
- $G_w$  the "warranted" rate of growth
- $x_0, x_1$  output, periods 0 and 1
- $s$  the savings rate as a fraction of income
- $C$  the "capital coefficient," "the value of the capital goods required for the production of a unit increment in output"
- $C_p$  the actual capital coefficient; "the value of the increment of capital per unit increment of output actually produced"

While  $G$  is simply the growth rate that actually occurs,  $G_w$ , the warranted rate "is taken to be that rate of growth which, if it occurs, will leave all parties satisfied that they have produced neither more nor less than the right amount. ...it will put them into a frame of mind which will cause them to give such orders as will maintain the same rate of growth."

The model that Harrod puts together from these concepts depends on two closely related equations. His "Fundamental Equation" is  $G_w = s/C$ . This gives the warranted rate. The formula for the actual rate of growth is  $G = s/C_p$ . The whole theory turns on divergences between these two.

unimportant. While he tells us that  $C$  can change, beyond vague references to "the state of technology" we are given no indication of how, when, or why it might do so. More importantly, in Harrod's actual description of the workings of the economy, he allows for no adjustments in technology. The only kind of adjustment his model allows to producing agents is a change in how much they produce – by the same technology. If, for example, producers find that they have not sold all of their output in one period, they respond by cutting back production in the next period; these cutbacks are general (because there are no different kinds of goods), and therefore the economy falls off Harrod's famous "knife-edge." What producers never do, in a Harrod world, is react to poor sales by improving their tools so that next year they can produce at lower cost and sell all their output by offering it at a more attractive lower price.

For Solow, the fixed relationship between capital and output is made quite explicit: "the capital/output ratio is ... constant – this is one of the defining characteristics of a steady state... (p. 33) Solow assumes constant returns to scale (p. 34), and contrives the model so that "technological progress augments labor only." (p. 35) That is, technological progress improves what human workers can do, not what their machines and devices can do. Increasing advances in productivity per person resulting from new capital goods are ruled out. In a Solow world there can be no fine new machines with which a company may halve its work force and still produce the same output.<sup>9</sup>

---

<sup>9</sup> This is true unless "the capital stock" can be "constant" even while the composition of that capital stock (to use a term Solow does not) changes. Solow suggests such a possibility:

The problem with an unchanging production function (let alone a function with constant returns to scale), is that it implies an absence of change in how things are done. But again, our inquiry is concerned with how we come to develop new tools and methods, which mean new and different ways of producing – a different "production function." Further, given the unfathomably complexity of the relationships among productive inputs, it would seem to be straining the metaphor to describe production as a function at all. It seems necessary, instead, to address directly the structural interrelationships among capital goods.

## 2.2. Missing structural elements: complementarity and indivisibility

Because it assumes that capital is homogeneous and unchanging except in quantity, the mainstream theory does not address fundamentally important structural aspects of capital which have been elucidated by the Austrian School, especially Ludwig

---

It should be realized that this reduction of technological progress to the efficiency-unit content of an hour of labour is a metaphor. It need not refer to any change in the intrinsic quality of labour itself. It could in fact be an improvement in the design of the typewriter that gives one secretary the strength of 1.04 secretaries after a year has gone by. What matters is this special property that there should be a way of calculating efficiency-units of labour, dependent on the passage of time but not on the stock of capital, so that the input-output curve doesn't change at all in that system of measurement. (p. 35)

The passage implies that improvements in capital can occur (e.g., the better typewriter) independent of a change "in the stock of capital." Surely this conception presents difficulties in how we measure the stock of capital, and invites the question of why technology which yields a better typewriter design is not "capital-augmenting."

Lachmann. A realistic view of the process of capital accumulation and its effects must take into account several factors that Harrod-Domar-Solow theory ignores.

The core point is that capital accumulation generally involves a lengthening of the capital structure, with what Lachmann calls a "division of capital," a specialization of individual capital items, which enables us to resist the law of diminishing returns" (1978, p. 79). Capital accumulation is primarily manifested not in the addition of more of the same. It occurs in what we might call a "complexifying" of the capital structure, an increasing in intricacy of the pattern(s) of complementarity among increasingly specialized capital goods, born in the on-going growth and division of knowledge.<sup>10</sup> Capital accumulation "does not take the form of multiplication of existing items, but that of a change in the composition of capital combinations. Some items will not be increased at all while entirely new ones will appear on the stage" (Lachmann 1978, p. 79). The homogeneity assumption obscures this key fact.

In pointing to "capital combinations," Lachmann stresses complementarity in this kind of process, and indivisibility of capital goods that is usually involved.

Generally the various items in a new, more complex capital structure have no usefulness at all except in combination with the other items, and those items are indivisible. "It will not pay to install an indivisible capital good," says Lachmann, "unless there are enough complementary capital goods to justify it. Until the

---

<sup>10</sup> Lachmann, following Hayek (1935), holds that over time there develops "an increasing degree of complexity of the pattern of complementarity displayed by the capital structure." (1975, p. 4)

quantity of goods in transit has reached a certain size it does not pay to build a railway" (p. 80).

A consequence of complementarities in capital use is that new economies of scale become possible, or rather economical, as a result of capital accumulation. These economies are the consequence not of the size of particular production processes (the sense in which we usually think of scale economies), but of the scope of their interaction. It makes sense to invest in a large-scale, indivisible capital item only in the presence of the necessary complementary capital. Lachmann gives a strong illustration: "The accumulation of capital does not merely provide us with the means to build power stations, it also provides us with enough factories to make them pay and enough coal to make them work" (p. 80). The greater scale economies possible in the power stations and the factories depend for their economic feasibility on one another. Similarly, it is said that the spreadsheet program drove the explosive sales of personal computers over the last decade – the tremendous economies that have been achieved in computer hardware over the last decade have been achieved through very large scale production, which itself has been driven by high-volume sales of popular software packages such as spreadsheets. On this view capital accumulation can affect growth in a way that is more exponential than geometric.

In Growth Theory, Robert Solow defines the stock of capital in his model as "the sum of past net investments" (1970, p. 4), maintaining the idea that new capital is simply added onto old. But because complementarity is fundamental to capital – because capital goods must be used jointly with some specific others – old capital is often destroyed in the process of capital accumulation; that is, its value is destroyed.

This is another basic fact of economic life that the Harrod-Domar-Solow approach ignores. Millions of dollars worth of whaling equipment was destroyed by the building up of the kerosene industry; vast quantities of iron-producing capital was destroyed by the advent of the capital goods that produce steel; software applications are made obsolete every few months as better come along. In the regrouping process that Lachmann describes, "some of these capital goods will have to be shifted to other uses while others, which cannot be shifted, may lose their capital character altogether. Thus the accumulation of capital always destroys some capital" (Lachmann 1978, p. 80).

Increasing returns to scale are also absent from the Harrod-Domar-Solow approach. Growing economies of scale are not inevitable, but likely in a free economy; they can and do result from the capital accumulation as it occurs in practice. (Young 1928) In Lachmann's terms,

We conclude that the accumulation of capital renders possible a higher degree of the division of capital; that capital specialization as a rule takes the form of an increasing number of processing stages and a change in the composition of the raw material flow as well as of the capital combinations at each stage; that the changing pattern of this composition permits the use of new indivisible resources; that these indivisibilities account for increasing returns to capital... (p. 85).

Theorists such as Harrod and Solow, and even Paul Romer, whose work we take up below, assume a diminishing marginal productivity of capital. This assumption would make perfect sense if the kinds of capital being used did not change, but because they do change, it makes no sense at all, not in consideration of the economy as a whole, over time. Because the capital structure improves, the tendency is to increasing marginal productivity of capital.

Again, while one can understand the desire of Harrod and his followers to simplify aspects of real world activity for convenience in their model, one must be wary of such simplifications as those made regarding capital. Simplifications which misrepresent and obscure do not aid understanding.

### 2.3. Shortfalls in the "new growth theory" of Paul M. Romer

In recent years, the theory of economic growth has been developed in what is known as the "new growth theory"; a major contributor to this literature is Paul M. Romer.<sup>11</sup> Romer brings up some of the issues with which we are concerned in this paper, and shows real insight into their importance.

#### **Valuable additional insights...**

While most growth theory has posited "given technology," or, where technological change is allowed at all, treated it as exogenous, recent work has dropped this assumption. Nelson and Winter (1982), for example, allow endogenous technological change into their evolutionary simulation model. Romer addresses endogenous technological change directly. Indeed, the title of a recent paper of his is "Endogenous Technological Change." (1990a) Among the premises of his argument which constitute new departures for growth theory are "that technological change – improvement in the instructions for mixing together raw materials – lies at

---

<sup>11</sup> See especially (1986, 1990a, and 1990b). Other important contributions include Lucas (1988) and Arrow (1962). For useful surveys of relevant work, see Diamond (1990), especially Dixit (1990) and Stiglitz (1990).



the heart of economic growth," and "that technological change arises in large part because of intentional actions taken by people who respond to market incentives."

Whereas Nelson and Winter retained the notion of homogeneous capital, Romer goes a step further, and explicitly includes heterogeneity of capital goods. "The unusual feature of the production technology assumed here," Romer says, "is that it disaggregates capital into an infinite number of distinct types of producer durables." (1990a, p.580)

Furthermore, Romer also brings out the link between knowledge and capital, ascribing the variety of capital goods to the different knowledge embodied in capital. He treats "long-run growth" as "driven primarily by the accumulation of knowledge by forward-looking, profit-maximizing agents," with a "focus on knowledge as the basic form of capital." (1986, p. 1003) This knowledge is embodied in capital goods:

The research sector uses human capital and the existing stock of knowledge to produce new knowledge. Specifically, it produces designs for new producer durables. An intermediate-goods sector uses the designs from the research sector together with forgone output to produce the large number of producer durables that are available for use in final-goods production at any time. (1990a, p.579)

Additionally, Romer takes seriously increasing returns in production where knowledge is increasing. His 1986 paper, entitled "Increasing Returns and Long-Run Growth," gives a "view of long-run prospects for growth" in which "per capita output can grow without bound, possibly at a rate that is monotonically increasing over time. The rate of investment and the rate of return on capital may increase rather than decrease with increases in the capital stock." (p. 1003)

In this work, then, we have reason to hope for some illumination about the relationship between capital goods and economic development.

**... but failure to develop the insights**

These hopes are disappointed, however. Romer seems not so much interested in exploring the implications of his insights as preoccupied with forcing those insights into the Procrustean bed of mathematical tractability. As a result, his treatment of capital and its role in production is still very meager. Indeed, his models take the life out of his introductory discussions.

Although Romer talks of and models technological change, the change he talks about is superficial. Consider the production function from the model in his 1990 paper:

[A] simple functional form for output is the following extension of the Cobb-Douglas production function:

$$Y(H_Y, L, x) = H_Y^a L^b \sum_{i=1}^{\infty} x_i^{1-a-b}$$

This production function differs from the usual production function only in its assumption about the degree to which different types of capital goods are substitutes for each other. In the conventional specification, total capital  $K$  is implicitly defined as being proportional to the sum of all the different types of capital. This definition implies that all capital goods are perfect substitutes. One additional dollar of capital in the form of a truck has the same effect on the marginal productivity of mainframe computers as an additional dollar's worth of computers. [This equation] expresses output as an additively separable function of all the different types of capital goods so that one additional dollar of trucks has no effect on the marginal productivity of computers. (1990a, p. S81)

$Y$  here is "final output";  $H_Y$  is "human capital devoted to final output";  $L$  is labor, and the various capital goods are the indexed values  $x_i$ .

To treat "output as an additively separable function of all the different types of capital goods" is to treat capital as homogeneous in fundamentally important respects, notwithstanding Romer's efforts to consider "distinct types of producer durables." Defining his production function in this way allows Romer to add additional types of capital goods indefinitely, just as Harrod could add additional numbers of capital goods indefinitely. In both cases, only the magnitude of the capital variable changes, not the form of the function. Implicitly, then, capital goods are all of a kind in respect to how they interact. To a given capital structure, add buggy whips or microchips (for Harrod add new quantities; for Romer, add new designs) and the effect on output will be the same, although the goods produced will differ. Capital is aggregable and thus implicitly homogeneous. Homogeneity of capital is further implied by Romer's construction of the production function as homogeneous of degree one (not so different from Solow after all). Where there are constant returns to scale, truly new and better production processes, which let us produce more with the same amount of input, are ruled out.

Lachmann's point that "[c]omplementarity is of the essence of capital use," (1978, p. 3, emphasis in original) is just as damaging to Romer's actual formulation of his model as it is to the work of Harrod. Romer leaves no room for complementarity, nor its concomitant substitutability (and hence capital destruction). In brief, Romer leaves no room for any of the structural aspects of capital that we will find to be of fundamental importance. To illustrate briefly, consider the relationships among three elements of the software capital structure: the programming system Smalltalk, the programming language COBOL, and WindowBuilder, a set of tools for developing graphical user interfaces. WindowBuilder is built in Smalltalk, for use with Smalltalk – without Smalltalk present it cannot work. COBOL is an older

programming language that is arguably being made obsolete by object-oriented languages such as Smalltalk. How are we to make sense of "additive separability" in respect to these three? Not only are Smalltalk and WindowBuilder directly complementary, in the strict sense that one requires the other to be running on the same computer, but also WindowBuilder itself, having been built in Smalltalk, could never have come into being without Smalltalk. Suppose we "subtract" Smalltalk from the equation, what becomes of WindowBuilder? Then it never was. These are not "additively separable." Furthermore, COBOL is being replaced by Smalltalk in certain cases. Then is the productive power of Smalltalk "added" to that of COBOL, or does it subtract from it?

In this work, we hold structural issues of complementarity and substitutability, as well as dependencies of one design on another, as of WindowBuilder on Smalltalk, to be of fundamental importance. We will find no help with these in the "new growth theory." Romer says, "An investigation of complementarity as well as of mixtures of types of substitutability is left for future work." (1990a, p. S81)

The main question this work seeks to help answer is, "What is the nature of the process by which people learn how to fashion better tools?" Here again, Romer gives little help. Within his broader model of a three sector economy, he models technological innovation as occurring in a research sector. He models the economy in three sectors. The research sector draws on available human capital and, making use of the current stock of technological knowledge, produces new technological knowledge in the form of designs for production goods. This new knowledge is then licensed to the production goods sector, which may build the designs into new and better capital equipment in subsequent periods. The new capital equipment is

then utilized by the final goods sector to produce consumable output. His substantive description of the process by which people learn how to fashion better tools is as follows:

It remains to specify the process for the accumulation of new designs, that is, for the growth of  $A_t$ . As noted above, research output depends on the amount of human capital devoted to research. It also depends on the stock of knowledge available to a person doing research.

Romer continues,

If designs were treated as discrete indivisible objects that are not produced by a deterministic production process, the production technology for designs would have to take explicit account of both integer constraints and uncertainty. There is no doubt that both indivisibility and uncertainty are important at the micro level and over short periods of time. The simplifying assumption made here is that neither is crucial to a first-pass analysis of technological change at the aggregate level. (p. S82)

After presenting an adjusted formalization of the model, he continues, "With this formal structure, the output of new designs produced by researcher  $j$  can be written as a continuous, deterministic function of the inputs applied." (p. S83)

Given our purposes, this is disappointing. Having been urged so far in the paper to recognize the importance of technological progress, we may naturally ask of it, "what is the nature of the process?" If so, we must content ourselves with the answer that technological progress is "a continuous, deterministic function of the inputs applied," that is, human capital and the stock of knowledge. It amounts essentially to this: when well-trained researchers are given a lot of good information, they think up new technologies.

The new growth theory has little to say about the process by which technological progress occurs. Indeed, it does not seem to be concerned with accounting for human economic advancement. Romer's paper does not; its attention is on requirements for and characteristics of a balanced growth equilibrium that is generated by the model as specified. Because it is based in a general equilibrium framework, there is no room for process: there is no uncertainty, no real time, no need for adjustment, no capital destruction. None of the richness of a mutual adjustment process in conditions of uncertainty is to be found here. The manner in which Romer formalizes his discussion takes the richness out of it, and leaves it in the end little better, for understanding the process of economic development, than the traditional models.

Like Harrod and Solow, Romer neglects the structural elements of capital. He chooses to ignore that the growth and division of knowledge leads to a growing complexity of complementary relationships among capital goods. For Romer, introducing new knowledge into production is essentially a research effort, not a coordination challenge.

### **3. Capital goods as knowledge**

To inform our examination of the process of capital development, we look in this section at capital itself. We find a fundamental relationship between knowledge and capital. Indeed, we regard capital as embodied knowledge of productive processes and how they may be carried out. Different varieties of knowledge are involved, as well as different kinds of embodiment.

### 3.1. Embodied knowledge

Carl Menger stresses the role of knowledge in human economic advancement: that knowledge is embodied in capital goods is fundamental to his thinking. 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) As this statement comes in a passage contrasting simple collection of first-order goods with employing goods of higher order in production processes, it is clear that we are to take the use of higher-order goods – capital goods – as the application of the knowledge Menger speaks of. When we know how to produce in a roundabout way, we employ capital goods for the purpose. Our knowledge is to be found in practice not in our heads, but in the capital goods we employ. Capital is embodied knowledge.

In particular, capital equipment – tools – embody knowledge, knowledge of how to accomplish some purpose.<sup>12</sup> Much of our knowledge of the causal relationships

---

<sup>12</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

between things, and of how to effect the changes we desire, is not articulate but tacit knowledge. In the beginning of 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 unskilled and clumsy academics to shave with the blade always at the correct angle, rarely nicking ourselves. How 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

---

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)



the embodiment of knowledge in a tool. The boy's observation and insight were built into the machine for use indefinitely into the future.

### 3.2. 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 such a form that they are 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 minerals (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.

With computer software, on which we will be focusing through most of this work, we have a logical extreme to inform further this approach to understanding capital goods. 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 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, and there are definite physical limits to computation. (Bennet 1985) Nevertheless, it is in the nature of computers and software to separate 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 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>13</sup>

Because the knowledge aspect of software tools is so clearly distinguishable from their physical embodiment, in investigating software capital we may distinguish clearly the knowledge aspects of capital in general. While software may seem very

---

<sup>13</sup> Personal conversation with Robert Polutchko of Martin Marietta Corp.

different from other capital goods in this respect, when we think in terms of the capital structure, we find 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 computer-assisted design (CAD) package is loaded in, the whole system becomes a dedicated writing or drawing tool.

But there is no important 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 hand tools are. When the knowledge of what is a hammer 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.

If this is true, then knowledge is the key aspect of all capital goods, because the matter is, and always has been, "there." As Bohm-Bawerk says in discussing what it means to produce:

To create goods is of course not to bring into being materials that never existed before, and it is therefore not creation in the true sense of the word. It is only a conversion of indestructible matter into more advantageous forms, and it can never be anything else. (1959, p. 7)

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<sup>14</sup>:

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.

### 3.3. 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. Perhaps it would be simple if knowledge were a simple, homogeneous something which could be pumped into a society as fuel is pumped into a tank. But knowledge is heterogeneous; it is not all of a kind. (Polanyi 1958, Hayek 1945, Lachmann 1986) There are important differences among different kinds of knowledge.

---

<sup>14</sup> (1981, p. 74). See also Vaughn (1990).

### **Articulate and inarticulate knowledge**

An important distinction in this respect is between articulate and inarticulate knowledge. Some of our knowledge we can articulate: we can say precisely what we know, and thereby convey it to others.<sup>15</sup> But much of our knowledge is inarticulate: we cannot say 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 possibly 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.

Furthermore, much of what we know we are not aware that we know. 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 do such and such," 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 have had time to think about it, and become explicitly aware, for the first time, of what we have long known. In this respect 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. He

---

<sup>15</sup> Those other, of course, bringing to our words different experience and outlook, will understand what we say somewhat differently from the way we do.

smiled and made the sound every child makes when imitating a fast car going past. Sure enough, the pitch goes from higher to lower – of course, I knew that; but I had not known that I knew it.

### **Personal and intersubjective knowledge**

There is also an important variety in what we may call the locations of knowledge. It may be internal – located within a person – or external, embodied in some intersubjective medium – located, as it were, among people, and therefore available for common use. In each of these locations there can be both articulate and inarticulate knowledge. I know my own verbalized thoughts and plans for the day, facts I learned in school, my phone number, etc. This is articulate knowledge in my own mind. Articulate knowledge can also be located externally, intersubjectively, in a form in which it can be transferred among people. This is the case with books, libraries, manuals, "for sale" signs, etc.

As we have seen, internal, personal knowledge may also be inarticulate. Most of our physical skills are of this kind. Our habits seem to represent a kind of inarticulate knowledge (in our habitual looking both ways before we cross streets is the knowledge that streets are dangerous), as do rules of thumb ('honesty is the best policy,' "get it in writing"), and social mores (waiting in line in crowded settings).

Extremely important kinds of knowledge are both inarticulate and external to individuals. Social institutions embody this kinds of knowledge. Language, for example, embodies a great deal of shared knowledge, accumulated over ages through interactions among people. As F.A. Hayek has stressed, there is a tremendous amount of knowledge in market prices. (1945) Don Lavoie has

developed this view (1985, Chapter 3), speaking of a "social intelligence" that emerges out of the interactions of people, which the society as a whole has, but no individual has.

*In this category of inarticulate knowledge located external to individuals, and thus available to be shared among individuals, is much of the knowledge embodied in tools. The crucial knowledge referred to by Menger above is of a kind we don't often think of as knowledge. It is not to be found in libraries or in books or in written words at all. Rather it is to be found in the designs of the tools we use. Much of it is inarticulate. Some may once have been articulated, but the articulation is now lost. Much may never have been articulate at all. Consider, for example, the ratio between the weight of a hammer head and the length of the handle. Hammer makers "know" the acceptable bounds of this ratio. How do they know? They know because the experience of generations has been handed down to them. Users of hammers, ages ago, found hammers with handles too long or too short to be uncomfortable; they discarded these and used the proper-sized ones instead. They could not have said why they did so -- they knew with their hands and arms, not with their heads. When they selected new hammers, they chose the ones with the "correct" ratio. From these choices hammer makers learned what the correct ratio was. The knowledge was gradually built into hammers over time, in an evolutionary fashion that depended on feedback from users. (Salin 1990)*

A significant proportion of the knowledge we use in production is not in any person or even group, but in the tools we use. I who use the hammer know nothing of ergonomics, and have not the slightest idea what the correct ratio of head weight to

handle length is. Nevertheless, when I drive a nail, I can tell if the hammer feels right. Thus I use that knowledge. The knowledge is built into my hammer.

### **Kinds of knowledge used in the production of capital goods**

We can distinguish three categories of knowledge that seem necessary in the development of new capital goods. Knowledge from each of these categories is embodied in every capital good.

#### **1. Knowledge of function**

The first category is simple: knowledge of what the tool must be able to accomplish – its function. What is this tool supposed to be able to do? Consider the development of the plow, for a simple example. Before plows can be developed, there must be farmers with the knowledge of what plows must do: turn over earth. Generally this function, whatever it might be, is only one part of a larger process involving other tools and processes. Accordingly, knowledge of function must include knowledge of the more encompassing production process of which this new tool will be a part.

#### **2. Knowledge of design**

The next category is more complex: knowledge of what style of tool might accomplish this – its design. Given the desired function, we need to know what kind or kinds of devices can accomplish that function. The farmer knows he needs earth turned over; now is needed knowledge of what kinds of devices, such as sharp, hard metal wedges, will turn over earth. This knowledge itself will be multi-faceted.



Because we are thinking in the context of a production process, this knowledge must comprise not only what that tool must do by itself (if, indeed, that has any meaning), but what might be contributed by other capital inputs so that the production process as a whole is successful. That is, what are the complementary capital goods and human capital (e.g. people's skills and techniques for effectively using such tools, as well as their habits and preferences) with which this sort of tool might work? Depending on the state of agriculture and mechanics, for the plow designer this might be knowledge of draft horses and harnesses, or of large tractors and hydraulics. The design of the plow will be influenced significantly by these complementarities, that is, knowledge of these complementarities must be built into the plow itself. The nature of the complementary goods will impose constraints on the design of the new tools, which must be made to fit. E.g., will the plow be attached to draft horses by leather harness, or to a tractor by a hydraulic yoke? Will the plow be guided all day by hand through the uneven turf, so that it must be light enough for a (strong) plowman to handle, or will hydraulics control the plow's angle of attack, so that a small boy with a good eye may direct a whole gang of plows from a tractor seat?

Not infrequently, important knowledge to be built into new capital goods will be not so much of extant complementary goods as of goods which are likely to exist by the time the new tool is actually produced. As technology continually leaps onward, tool designers seem to plan their products in an anticipatory fashion: they design new production goods with an eye to the necessary complementary goods that seem likely to become available. The longer the period between design and production, the more aggressively it makes sense to anticipate. We see this anticipation clearly in the software industry. Large applications with demanding

speed and memory requirements are designed well before such speed and memory are affordable to the software's target audience. The software developers are willing to plan so aggressively in the expectation that the price of processing power and memory will continue to fall at rapid rates.

A valuable element of design knowledge that may be brought to the design of a new tool is knowledge of how to make the design itself readily adaptable, so that when the inevitable changes in conditions and complementary goods occur, it will be relatively easy to alter the design as necessary. This is characteristic of what we might call effective flexibility. We will have much to say about it in Chapter 4.

### 3. Knowledge of construction

The third category is knowledge of how to construct such a tool, how to effect that design – its construction. The designer might see that his design will serve the purpose called for, and yet not know how to build what he has designed. I might design a baseball bat, for instance, specifying the kind of wood to be used and laying out the exact shape in a drawing. But I don't know how to use a lathe. Before the bat could come to be, another kind of knowledge than mine is required: the knowledge of how to take a fully worked out bat design and embody that in actual wood. The plow designer might specify precisely the shape and hardness of the blade, yet have no idea of how actually to produce it. The actual plow must ultimately embody also the knowledge of those who operate (and of those who designed) the forge in which the plow is cast. Implicitly, then, construction knowledge comprises knowledge of the higher-order tools, as well as raw and intermediate inputs, with which the new tool might be built.

Of course there is often a lot of overlap between knowledge of design and knowledge of construction. Designers generally need to be cognizant of what construction techniques are available, and often the availability of newer and better techniques will inspire and inform new kinds of designs. But the two kinds of knowledge are categorically different. Indeed, it is possible to design things which cannot be built, given the present state of materials and engineering technique. (Drexler 1991) We might design a variation on a spider web, for example, specifying spider's silk as the construction material. While there would be no ambiguity in the design, that design could not be built, because humans cannot yet produce spiders' silk. One might design a plow, say, one tenth the weight of current steel plows, and ten times harder and stronger, but such a design could not yet be built because we do not know how to produce such a material.

#### 3.4. A subjectivist view of capital

Before going further, let us set out more explicitly what is meant by capital. In keeping with the tradition of the Austrian School, we take a subjectivist viewpoint, and insist that to be capital, something must be treated as capital, that is, treated as some kind of input in the production process. Lachmann writes,

Beer barrels and blast furnaces, harbour installations and hotel-room furniture are capital not by virtue of their physical properties, but by virtue of their economic functions. Something is capital because the market, the consensus of entrepreneurial minds, regards it as capable of yielding an income. (p. xv)

Something need not be physical to be capital. An obsolete railroad engine, though blatantly physical, is not capital once it is abandoned and forms no part of any production plan. On the other hand, something without physical properties, such

as a set of sound construction processes, or a successful design type, or the experience of skilled designer or craftsman, is capital because it is used (consciously or unconsciously) as an input in the production process. Something is capital insofar as it is an input into a production process. Hence knowledge can be capital if it is treated as a (scarce) input in a production process.

As Bohm-Bawerk said, capital is "the produced means of production."<sup>16</sup> "Produced" suggests some directed activity to accumulate the knowledge as a means of production. Bohm-Bawerk defines capital as "nothing but the sum total of intermediate products which come into existence at the individual stages of the roundabout course of progression (sic; "production"?)." (1959, p. 14) Again, "intermediate product" suggests something produced, something intended to be produced, as an intermediate good. Knowledge produced for use in production is capital.

### 3.5. Varieties of embodiment of knowledge

For knowledge to be capital, it must be usable in production. Accordingly it must be "stored up" in some sense, embodied, brought together in a form in which it will be more or less handy, ready to use in a production process. There are a number of ways in which this knowledge may be embodied.

As the language we use here is potentially misleading, let us take a moment to clarify: Capital is embodied knowledge; yet it need not have any physical aspect:

---

<sup>16</sup> But as Lachmann points out, "the question which matters is not which resources are man-made, but which are man-used." (1978, p. 11)

the knowledge need not be embodied in any physical body. By embodiment we mean a metaphorical embodiment. Knowledge becomes capital as it is sorted out and "put away" somewhere where it will be ready to hand – available and ready to be used – when production time comes along. By embodied, we mean synthesized, localized, put in order, focused in a manner that will make it usable in the anticipated production process. Wherever, or in whatever, the knowledge is "put away," that is the thing in which it is embodied.

*Knowledge may be embodied only in tacit form, in people's orientations. Standard practices and rules of thumb fall into this category. It may also be embodied in persons' minds and motor nervous systems. This is human capital – background knowledge, familiarity, skills and experience.*

Knowledge may be embodied in texts of some kind – symbols largely unconstrained by physicality. In this category are procedures, software, recipes. The medium to which the text is written is quite independent of the text itself. E.g., the maintenance procedure for a machine may be posted on the wall above the machine, or kept in the minds and habits of the foreman and machine operators. A favorite recipe can be written down, or kept in one's head. A computer program is essentially the same whether typed on a computer's screen editor, printed out on paper, compiled into executable form in the circuits of some particular computer, or stored on a disk or tape. What is important is that it is embodied in some stable medium, accessible to a number of people, so that it may be used.

And of course the knowledge may be embodied in materials, as in our examples above of the plow, the hammer, and the computer program actually loaded into a computer's circuits and running.

#### **4. Capital goods and division of knowledge across time and space**

There is a distinctly social nature to capital goods, and the capital structure which they compose (along with a host of supporting institutions and shared cultural understandings). Most individual capital goods are manifestations of a far-flung division of knowledge, an almost incomprehensibly extensive sharing of knowledge and talent across time and space. The ever-changing pattern of the interactions of these capital goods – the capital structure as a whole – is certainly beyond our grasp. It is a part of what Hayek called "the extended order of human cooperation." To pursue further this idea that capital goods and the capital structure manifest a profound social interaction, let us consider Adam Smith's discussion of the division of labor, to which he attributed the lion's share of human progress.

Recall Smith's case of improvement to the steam engine, which grew out of a small boy's observation that he could tie a piece of string from the handle he was assigned to operate to another part of the machine, and so get the action of the machine to do his job for him. Subsequently this insight was built into the design of steam engines. When, in cases such as this, knowledge is built into a piece of capital equipment so thoroughly that an actual person is no longer required, what has happened to the division of labor? Has it decreased? The little boy is no longer at work at the steam-engine. Does his departure diminish the division of labor present in that production process? Are there, in a sense, fewer people contributing?

It appears that what Adam Smith meant by the division of labor was the division, among a number of different people, of all the tasks in a particular production

process. Given a number of tasks which are visibly part of the production process, the fewer the instances in which the same person carries out more than one of those tasks, the greater the division of labor. This view is evident in Smith's remarks on agriculture:

The nature of agriculture, indeed, does not admit of so many subdivisions of labour, nor of so complete a separation of one business from another, as manufactures. It is impossible to separate so entirely, the business of the grazier from that of the corn-farmer, as the trade of the carpenter is commonly separated from that of the smith. The spinner is almost always a distinct person from the weaver; but the ploughman, the harrower, the sower of the seed, and the reaper of the corn, are often the same. (1976, pp. 9-10)

Here Smith focuses on division of labor among those directly involved in a production process: how many laborers are involved at that time and place, given the tools they have. We take issue with Smith, holding that the division of labor is better understood as the whole pattern of cooperation in production, direct and indirect. The indirect contributions are, in an advanced economy, the most significant. As Carl Menger pointed out, the crucial "labor" is the creative effort of learning how,<sup>17</sup> and the embodying of that learning in a tool design that can be used by others, who themselves lack the knowledge in any other form. We really do better to speak of the division of knowledge rather than the division of labor.

Axel Leijonhufvud makes clear the importance of the division of knowledge in his article, "Information Costs and the Division of Labor" (1989). He invites us to consider a medieval serf, named Bodo, and asks "Why was he poor?" Leijonhufvud

---

<sup>17</sup> (Menger 1981). For a discussion of Menger's criticism, see Vaughn (1990).

argues, "Bodo was poor because few people co-operated with him in producing his output and, similarly, few people co-operated in producing his real income, i.e. in producing for his consumption" (p. 166). The cooperation need not be on the same spot and at the same time to be relevant. Indeed, as an economy advances, the pattern of cooperation spreads out spatially and in time.

Our rich twentieth century representative man, then, occupies a node in a much larger network of co-operating individual agents than did poor Bodo. His network, moreover, is of very much larger spatial extent. The average distance from him of those who contribute to his consumption or make use of his productive contribution is longer. Similarly, his network also has greater temporal depth – the number of individuals who † periods into the past made a contribution to his present consumption is larger than in Bodo's case. (Leijonhufvud 1989, p. 166)

In his comments on the division of labor in agriculture, Smith neglects the division of knowledge and of labor implicit in the tools the farmers use. The plough, the harrow, and the scythe (or in our day the John Deere combine<sup>18</sup>), themselves represent an extensive division of labor and, more importantly, of knowledge. To be consistent with his suggestion in the quoted passage, Smith would have to assert that there is less division of labor represented in the present day manufacture of pins, in which (if I guess correctly) hundreds of thousands may be made in a day in a fully mechanized process overseen by one technician at a computer terminal, than in the factory of which he wrote. But the fact that there is now only one person there on the spot does not mean there is no division of labor in pin-making. It illustrates, rather, that the division of labor is now more subtle: it is manifested not

---

<sup>18</sup> ...which reaps scores of acres in hours, while its driver sits in air-conditioned comfort listening to Willie Nelson in stereo.



in many workers, but in very sophisticated tools to which many creative workers have contributed their special knowledge of the steps (what used to be the tasks) involved in pin-making. Today's equivalent of Smith's division of labor is manifested in a complex division of knowledge embedded in a deep pin-making capital structure.

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 capital structure, each of us is able to take advantage of the specialized knowledge of untold others who have contributed to that structure. The structure becomes increasingly complex over time, as the pattern of complementary relationships extends.<sup>19</sup>

In capital-intensive, modern production processes, the division of knowledge and labor is to be found not in the large number of people at work in a particular production process, but in the tools used by a very few people who carry out that process. The knowledge contribution of multitudes is embodied in those tools, which give remarkable productive powers to the individual workers on the spot. The little boy is there in a modern steam engine, his knowledge embodied in the valve-control rod. The farmer at his plough is empowered by the knowledge and labor of hundreds of others, who designed his plough and hardened its steel, who developed his tractor, who learned how to refine its fuel, etc.

---

<sup>19</sup> Lachmann credits Hayek (1935) with "reinterpreting the extended time dimension of capital as an increasing degree of complexity of the pattern of complementarity displayed by the capital structure." (1975, p. 4)

The point is emphasized by Bohm-Bawerk, who in the following passage could be responding to Smith's above comments on agriculture:

...the labor which produces the intermediate products ... and the labor which produces the desired consumption good from and with the help of the intermediate products, contribute alike to the production of that consumption good. The obtaining of wood results not only from the labor of felling trees, but also from that of the smith who makes the axe, of the carpenter who carves the haft, of the miner who digs the ore from which the steel is derived, of the foundryman who smelts the ore. Our modern system of specialized occupations does, of course, give the intrinsically unified process of production the extrinsic appearance of a heterogeneous mass of apparently independent units. But the theorist who makes any pretensions to understanding the economic workings of the production process in all its vital relationships must not be deceived by appearances, his mind must restore the unity of the production process which has had its true picture obscured by the division of labor. (1959, II, p. 85)

What a difference there is between the meaning Bohm-Bawerk attaches to the division of labor in this passage and the view suggested by Adam Smith in his comments on agriculture. For Bohm-Bawerk, the division of labor is extended down time and across space. The miner of the ore is "there," in a sense, as the lumberjack fells trees with steel made from that ore. In an advancing economy, the division of knowledge is an ever-widening system of cooperation in which are developed new tools and processes whereby each person may take advantage of the knowledge of an increasing number of his or her fellows. The division of knowledge is manifested in the tools we work with, which embody the knowledge of many.

## 5. Capital structure

Capital exists and works within a structure. (Lachmann 1978, Hayek 1941) It is an ever-evolving structure to be sure – it is never static – but at any time the relationships among capital goods, and among capital goods and human capital, are essential. Of the various perspectives we might take on the capital structure, three will be important to us. One looks at the relationships of complementarity between capital goods used jointly in a production process; another looks at relationships of dependency between capital goods, one or more of which are used in producing another; a third looks at the different categories of capital which are involved in production processes.

### 5.1. Complementarity of the essence

We have said enough already of the importance of complementarity so that we need not discuss the point at length. Let us merely reemphasize it. Lachmann says,

It is hard to imagine any capital resource which by itself, operated by human labour but without the use of other capital resources, could turn out any output at all. For most purposes capital goods have to be used jointly. Complementarity is of the essence of capital use. But the heterogeneous capital resources do not lend themselves to combination in any arbitrary fashion. For any given number of them only certain modes of complementarity are technically possible, and only a few of these are economically significant. (p. 3, emphasis in original)

Programming languages run only on certain kinds of computers. A complex programming environment such as the object-oriented system Smalltalk requires further that the computer be equipped with a mouse, and a high-resolution display. The various graphical user interface builders for Smalltalk run only where certain

specific versions of Smalltalk are present. These are very powerful tools, but usable only if the necessary complementary goods are present.

We will devote a whole chapter, Chapter 4, to the subject of capital maintenance. In the present context it is important to point out that the challenge of capital maintenance has fundamentally to do with complementarity. Capital exists and functions in a structure in which complementarities are fundamentally important, and the capital structure evolves over time as old tools and processes are supplanted by new. Consequently, for any particular (kind of) capital good, maintenance is very much a matter of maintaining its complementarity to the rest of the changing capital structure. Hence maintenance may mean not only preventing any change through deterioration, but actually changing that (kind of) good directly, in a manner that adapts it to the changing capital structure around it, and thereby delays obsolescence.

Because change is pervasive, how a particular (kind of) capital good is used will inevitably change. As Hayek has pointed out, (1935) capital maintenance is often more a matter of maintaining the value of capital than merely preventing decay. But because value depends on position in a changing capital structure, maintaining value may mean changing the good more than preserving it as is.

Software, of course, does not deteriorate. (A diskette may, but a diskette is software's storage medium, not software itself.) Yet programmers speak of "bit rot," that creeping incompatibility that erodes software's usefulness as the environment changes – with new computers, peripherals, operating systems, etc. – and the code does not. This is purely a matter of complementarity. To maintain the value of a piece of software, even when what it does stays exactly the same, requires changing

that software to keep it complementary to the changing capital goods with which it must work.

The point applies to capital goods generally. As tractors replace horses and oxen, plows must be equipped with different attachments, and ganged two, three or more abreast to take advantage of the greater power. As microwave ovens become popular, some kinds of cookware must be made microwave-safe. There may be a great deal of consistency in essential features of the designs: the geometry and hardness of the plow blades, and the shape, weight, and appearance of the cookware may remain the same. But if the plow and cookware makers are to stay in business, if their products are to be valued and used in the newly-evolved production processes, then they must be altered appropriately. To maintain the value of different (kinds of) capital goods is to change them as necessary to maintain their complementarity to the evolving capital structure in which they play a part.

## 5.2. Orders of capital goods

It is useful to think of capital in terms of orders of goods,<sup>20</sup> consumer goods being goods of the first order, and capital goods being goods of higher orders. As the capital structure lengthens, we develop tools for producing tools for producing tools... The better the tools at each stage, the better and more cheaply we may produce the goods at the next lower stage. Menger stressed the importance of lengthening the capital structure:

---

<sup>20</sup> See Mises (1966, pp. 93-4)

Assume a people which extends its attention to goods of third, fourth, and higher orders... If such a people progressively directs goods of ever higher orders to the satisfaction of its needs, and especially if each step in this direction is accompanied by an appropriate division of labor, we shall doubtless observe that progress in welfare which Adam Smith was disposed to attribute exclusively to the latter factor. (p. 73)

Improvements in tools (and related processes) of high order are very important to economic development, because those improvements can be leveraged throughout the production process.<sup>21</sup>

Frequently, there is a kind of recursion involved, in that developments at one stage make possible developments at another stage, which can in turn improve processes at the first stage. Better steel, for example – the product of a steel mill, makes possible the construction of better steel mills. The availability of the language Smalltalk made possible the user interface builder WindowBuilder, which is itself an improvement to Smalltalk.

We will be interested in most of what follows with goods of fairly high order, in particular, with in large part with tools for the design of tools. To clarify this point, we need to consider the different categories of capital inputs to a production process.

---

<sup>21</sup> The very fact that there are orders of capital goods calls into question Romer's assumption that new capital goods have an additively separable effect of output. There is always dependency of lower order goods on the higher order goods that produce them; hence treating these goods as separable in their effects is nonsensical.

### 5.3. Categories of capital goods

What are the categories of capital goods at work in production processes? We will distinguish first between fixed capital: "producer durables" such as tools and machinery; and working capital: raw materials or intermediate goods, or goods in process. Examples come readily to mind when we envision a production process. In the steel mill the mill 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 a business context, we might think of the word processor and spreadsheet program as the fixed capital, and a company's raw data as working capital, to be processed by the spreadsheet into, say, a meaningful report.

But this capital does not work by itself. In order to be productive, it must be put in motion and directed by people according to some plan, in a set of procedures. Accordingly, to our list of categories we add procedures. These three are inextricably interrelated, because the procedures will be couched in terms of what the tools do to the materials. You can't have procedures without the other two. These procedures can be stored (embodied) in a variety of ways, e.g., in written documents, in the "human capital" of a skilled worker's mind, muscles, or senses, in machines which embody them (as a grain combine combines cutting and threshing in sequence), and even in rituals.

An illuminating example of a procedure stored in a non-material fashion is that of the ritual of sword-making in ancient Japan:

[T]he techniques that produce the special properties of steel ... reach their climax, for me, in the making of the Japanese sword, which has

been going on in one way or another since AD 800. the making of the sword, like all ancient metallurgy, is surrounded with ritual, and that is for a clear reason. When you have not written language, when you have nothing that can be called a chemical formula, then you must have a precise ceremonial which fixes the sequence of operations so that they are exact and memorable. ...

The temperature of the steel for this final moment [when it is plunged into water to cool] has to be judged precisely, and in a civilisation in which that is not done by measurement, "it is the practice to watch the sword being heated until it glows to the colour of the morning sun."  
(Bronowski 1973, pp. 131-33)

Some kinds of computer programs embody procedures, e.g. those that direct automated assembly on an assembly line.

Fixed capital, working capital, and procedures – is that all? No. These three imply some purpose, some end being aimed at. Our procedures for applying tools to raw materials aim at producing something. This something must be conceived, more or less fully. To put it another way, it must be more or less fully designed. So the producers in a production process must have some implicit or explicit design to inform the whole process. This design, this conceptualization or description of what is aimed at, is what guides the procedures.

In this category are sketches and detailed blueprints and specifications, CAD pictures in all the range of possible detail, vague mental pictures, detailed models, software prototypes and completed code, and generally accepted definitions. Examples are "steel rail," which design (probably in the form of a detailed specification) informs the procedures of the steel mill, "loaf of bread," which informs the procedures of the baker, and some notion of a report on profitability projections, which informs the procedures of the business analyst.



Thus we have four elements of production processes: 1) tools or fixed capital, 2) raw or intermediate material, or working capital, 3) procedures for applying the tools to the raw or intermediate goods, and 4) designs which inform the procedures.

## **6. Capital development as a social learning process**

Consider the implied context of the above discussion. We spoke of production processes, implicitly, of known production processes aimed at producing known goods. The designs of which we spoke, which inform the procedures directing fixed capital in processing working capital, are themselves implicitly known. But this begs an important question: where do the designs come from? How are they produced? What is the process by which they come to be?

It is important here to draw a clear distinction between producing designs for goods, and producing individual instances, real cases, of those designs, because the production processes are different. And, living as we do in a physical world, where physical instances catch our eye, it is easy to overlook the production of designs, and see only the production of instances. Economics, certainly, has overlooked the production of designs, by and large assuming it away: standard models assume "given technology" or use of the "best available technology." But for our purposes - - investigating how the capital structure develops and improves - it is essential to focus on production of designs as an activity different from the production of particular goods embodying those designs.

Let us clarify the distinction by contrasting 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 Microsoft's work producing software, by contrast, we think of programmers writing code – creating new designs (or enhancing older designs). True, Microsoft employs 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>22</sup> Not at all. In fact, we hold that 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 that design of capital goods and what we will call their instantiation – the creation of actual instances of those designs – are fundamentally different from the analytic viewpoint. In practice we cannot always separate the two, because design and instantiation frequently occur simultaneously, but in principle they are different kinds of activities; they aim

---

<sup>22</sup> Indeed, product design in manufacturing industries is receiving a lot of attention. See Wheelwright and Clark (1992), and Womack et. al. (1990).

at different goals. Design is concerned with the known, instantiation with the unknown. Design is a matter of bringing together knowledge of how to accomplish productive purposes that has not yet been brought together in that manner; instantiation is a matter of imprinting a design onto a different medium. To design a capital good is to work out fully what it should be, to instantiate such a capital good is actually to bring it into physical being.

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.

What is the nature of this process? What makes it go forward better or worse? We turn now to an examination of software development, in order to find some answers to these questions.

## Chapter 2

### A Short History of Software Development

*Yet I doubt not through the ages one increasing purpose runs,  
And the thoughts of men are widened with the process of the suns.  
- Tennyson, "Locksley Hall"*

*... it's taken us years to understand just how hard it is to build good software. Developing robust, large-scale software systems that can evolve to meet changing needs turns out to be one of the most demanding challenges in modern technology.  
- David Taylor<sup>23</sup>*

#### 1. Introduction

The goal of the remainder of this work is to understand better the manner in which the capital structure expands and improves. Our contention is that the most important characteristic of capital for growth and development is the knowledge embodied in the things we think of as capital. We study software because software is a kind of capital good in which knowledge is peculiarly evident. Software is almost pure knowledge. With software it is easy to see the distinction between design and instantiation that is inherent in all goods. Designing software – bringing together the relevant knowledge of how a computer may be programmed for some purpose, and embodying this knowledge in code – is challenging. In this, software is like other goods: designing effective capital goods is of any kind is challenging.

---

<sup>23</sup> (1990, p. 2)

In regard to instantiation, however, software is very different. Instantiating a program – creating an instance or another copy of it – is utterly simple. It can be done in microseconds, with a couple of keystrokes. It requires no factories, no steel or glass or plastic. (Indeed, one of the obstacles to vigorous markets in software capital, as we will see in Chapter 5, is the challenge of establishing and defending property rights to goods that can be copied at virtually no cost.) Because with software this crucial knowledge aspect is so distinct from the physical, by studying software development we can focus on the knowledge aspects of capital and capital development without distraction. By studying software development, then, we hope to learn more about how the capital structure in general expands and improves.

Because economists who read this work may be unfamiliar with software development, at this point we devote a short chapter to orienting those readers with a brief historical overview of software development processes and tools, and how they have evolved. The chapter is essentially an introduction to the empirical part of the dissertation for those unfamiliar with computer programming. It introduces the main concepts that will, in subsequent chapters, be elaborated and related to capital theory and issues of economic development.

We first consider the main forces that have driven the evolution of programming practice. Foremost among these is the astonishing fall in prices of computer processing power and memory, which has enabled ever larger and more ambitious programming projects. Bringing these projects to fruition has not been easy. The main challenge, which we take up next, has been managing the software's complexity. Doing so is difficult, and a variety of tools and software development

methodologies have been put into practice to try to meet this challenge; we take an overview of these. We finish by introducing object-oriented programming systems (OOPS) and related technologies, a relative newcomer to the field that seems to hold real promise for enabling coordination in these complex capital structures.

## **2. Overview**

As Lavoie, Baetjer, and Tulloh (1992) have pointed out, the evolution of programming practice seems to have been driven by the steady drop in the price of computational resources. As processing power, memory, and storage space have dropped dramatically in price, people's software ambitions have grown apace. Once programming was mainly resource-constrained: with processing power and memory scarce and expensive, our programs were necessarily simple, and programmers concentrated on making the most of the scarce machine resources. After all, if a program was too big, it might not fit into the computer; if it was not very cleverly executed, it would take prohibitively long to run. But the resource constraint has been relaxed by the prodigious productivity of hardware manufacturers. As a result, the programs we have tried to build have become more and more ambitious and complex. We can afford – in respect to memory requirements – to build big programs because memory is cheap. We can afford – in respect to processing power – to demand a tremendous amount of computation because our machines are so fast. In short, we can afford – in respect to physical resources generally – very big, very complex programs. Accordingly, we try to build such programs, sometimes with success, sometimes without. But building

larger programs necessitates coping with increasing complexity on a number of areas.

One source of increased complexity arises from the very division of knowledge on which major software projects depend. As the software industry has grown in size and ambition, software development has of necessity become less and less a solitary activity, and more and more a group endeavor, with many people contributing their knowledge and talent to the development of a software system. Large projects cannot be completed in reasonable time by a single person, especially where a variety of specialized capabilities, each depending on quite extensive domain knowledge, must be incorporated. Hence team programming. It is not unusual to have several hundred programmers all working on the same project. The different programmers are often separated both geographically and in time, as they work on different parts of a large system in different locales and on different schedules.

Another source of greater complexity is the integration of various functions into one software system. There was a time when each application stood more or less alone. Now, however, we want our different software tools to "talk to" one another – we want them to complement one another. A simple example is the integration of word-processing, spreadsheet, and graphics capabilities: modern word-processors import drawings, charts, and tabular data from other programs. A different kind of integration is the "embedding" of software into physical machines. "Embedded systems" direct machines, sensing and controlling, for example, movements of robot arms, temperatures in ovens, and the flow of inventory through a manufacturing process.

Still another source of complexity is networking. Programs were once confined to the computer that they ran on. Now, with improved telecommunications and computer networking, computation has become very much a social process. It is increasingly inapt to say that certain programs runs on "a computer." Frequently, they run on several machines at once, their functionality extended across the network, with many people interacting through them; such applications are known as distributed applications. For example, automated teller systems interlink a host of different automated teller machines at many different sites, serving many different banks. In the new world of distributed applications, it is said that "the network is the computer."

All this increased complexity is problematical. In short, as we have made great progress in overcoming the resource constraint on programming, we have encountered a complexity constraint.

### **3. The Key Challenge: Managing Complexity**

The primary constraint on programming today is not physical resources, but the limits of our ability to manage complexity. As one software designer puts it, the key limitation is "our sheer ability to understand what it is we are trying to do."<sup>24</sup>

As programs grow in magnitude and complexity, division of knowledge becomes a necessity. (Lavoie, Baetjer, and Tulloh 1991a) There is a limit to how much code one person can keep in mind and work with at one time, so the task must be split

---

<sup>24</sup> Mark S. Miller, personal conversation.



up somehow. Merely to get a grasp on what is happening, we have to abstract from the whole problem, decomposing it somehow into subsystems and subproblems (of succeeding levels) which different people may work on, or the same person at different times. This decomposition occurs in various ways, some of which we will examine below. But one way or another, large programs must be split up into different modules, in order to allow the programmers to focus on the different parts of the problem. "This general strategy is known as modular programming, and it forms the guiding principle behind most of the advances in software construction in the past forty years." (Taylor 1990, p. 3) How the abstraction boundaries are drawn is important. Appropriate abstractions provide order and understandability; inappropriate abstractions cause problems.

A major problem is incompatibility among modules. Even when only one person is working on a complex problem, it is easy to forget, or simply to misunderstand, the effect that one module may have on another, and thus to build in unwanted (side) effects – bugs. Much of debugging a program has been to this point a matter of ironing out all these unintended interferences of one module with another. As we will see, better-conceived ways of drawing abstraction boundaries can significantly diminish this discoordination.

Of course large projects are often undertaken by large groups of people, with a different person or team working on each module, and with a system architect or system designer overseeing development at a high level. With this division of knowledge not only among different modules but also among people, there arise additional coordination problems. These have been explicated well by Fred Brooks in his celebrated book, The Mythical Man-Month (1975). Brooks emphasizes the

importance of communication: the different team members must keep informed of what assumptions being made by others, which will affect what they themselves are working on. There can be great difficulty in maintaining effective communication and clear understandings among the members of a team when the team grows large: at some point the sheer cost of maintaining effective communication exceeds the value of the additional manpower.

#### **4. The Evolution of Programming Practice**

In response to the challenge of managing the ever-increasing complexity of software, the software industry has evolved a set of higher-order capital goods and corresponding practices to help them build knowledge into software in an orderly, effective way. These include new programming languages and a variety of tools and processes. They continue to evolve rapidly.

##### 4.1. Programming Languages

A primary aid to managing complexity is the development of higher-level programming languages. Each new generation of languages gives programmers increasing power to express complex relationships by capturing and expressing higher-level abstractions. Each gives programmers more freedom from the concerns of the computer itself – such minutia as what value is in what register – enabling them to think more in terms of the problem they are trying to solve and less in terms of how the computer operates to solve it. In the earliest days, on machines such as ENIAC, "programmers" actually twisted dials and moved connector cables on the machine. In place of this physical manipulation now there is machine language.

Up a level of abstraction from this is assembly language, still highly numerical, concerned with the needs of the machine. Gradually, as one passes to higher and higher level languages, the code becomes less oriented to the characteristics and needs of the machine and more attuned to humans' characteristics and thought processes. Accordingly, programmers using these languages can think in terms of familiar words which represent aspects of the problem domain they are trying to represent, unconcerned with the details of how a particular machine will store and manipulate bits and bytes.

At the same time higher level languages provide better abstraction capability, they provide more discipline – and hence understandability and coherence – to the code. Somewhat paradoxically, languages which provide programmers great freedom provide them with the rope to hang themselves. Programmers learned early to write subroutines – sequences of instructions treated as separate units, which can be called from anywhere in a program – to which they directed program flow with GOTO statements. But the unrestricted use of GOTO statements leads to "spaghetti code," in which the relationships among different modules are difficult or impossible to perceive, making life difficult for anyone, including the original programmer, who might come back to this code to work on it. There is a tradeoff, in programming, between flexibility and manageability. Languages and techniques which allow great virtuosity also allow code to be made incomprehensible. Languages and techniques which limit also discipline, and thereby lead to more understandability.

Structured programming languages respond to this tradeoff by providing programmers a relatively small but comprehensive set of functions for directing

program flow, so that the underlying structure of the program is much more clear and understandable. But structured programming languages still share a common pool of data. While the functions that the program performs are separated into clearly-structured, separate routines, all the data that the program uses is centralized and accessible to any of those routines. As a consequence, one routine too frequently changes data structures in a manner not anticipated by other routines, leading to nonsense – bugs.

A recent response to this difficulty (and others) is object-oriented languages. Because these seem to constitute a fundamental change of approach, we will take them up in some detail in the last section of this chapter.

#### 4.2. Development methodologies

Along with programming languages have evolved various software development methodologies. A methodology is set of procedures that a software development organization follows (or tries to follow) in producing new software. Again, in the early days, when computers were very limited and problems were relatively simple, no extensive methodology was necessary. Good programmers could "hack" a solution, working at the problem in an unstructured way until they solved it. But as programs grew, this approach broke down. It became impossible to predict when a program would be ready for use, whether or not it would work properly, and, if it did work, whether or not it would be what the customer actually needed.

There grew up in response a move to discipline the software development process, to make it more like other kinds of engineering in being based on sound, established principles and "industry-standard" processes. Hence the term "software

engineering." Whether because of the youth of the industry, or because of the special nature of software, industry-standard processes, with resultant standardization and predictability, have decidedly not emerged. The whole field of software development methodology remains in ferment, with new methodologies growing up amid high hopes, and then fading in disappointment. There is a coevolution of software development tools to support the various methodologies, which we consider below; and because the technologies, needs, and tools of the industry are changing so rapidly, there is little stability or accepted wisdom. Software development is difficult. As Fred Brooks wrote in the title of a celebrated essay on the subject, there is "No Silver Bullet" with which to slay the problems and make software development easy. (1987)

Nevertheless, attempts must be made, and they show some success. Traditional methodologies generally consist of some variant of the "waterfall model," in which development cascades from users' requirements to analysis to design to coding, to testing, to debugging, to delivery. Such methodologies are a reaction from the unstructured, experimental approach of the early days. Often they are associated with special tools called CASE tools (see below), CASE standing for computer assisted software engineering. These approaches are sometimes known as CASE methodologies. In an attempt to bring the rigor of engineering to software development, these methodologies aim to make clear at the outset exactly what the user wants; this is the requirements stage. There follows an analysis of the problem domain and the physical environment (e.g. computer types and network needs) in which the software will run. Then there is a high level design of the system. The analysis and design are frequently captured in complicated drawings of data flows and entity relationships. Next the coding is done; frequently this is a matter of

translating the elaborate design drawings into code. Then the code is tested and debugged and finally, one hopes, delivered to a satisfied customer.

As we will see, these traditional methodologies have fallen short of what was promised for them, often because they assume that requirements can be clearly established at the outset of the development process. Because the knowledge which must be built into software is dispersed and tacit, it is rare to get a clear, complete statement of requirements, especially in recent years. In the early days of electronic computation, computers were used mostly to automate well-known, established processes. Hence the task of the software was reasonably clear. But as programmers became more sophisticated, and as people gained experience in using computers, they began to try to take advantage of computers in new ways, not just doing the same old thing faster and cheaper, but doing something new, different, and better. Requirements for such systems cannot be stated clearly at the outset, because people do not know yet what they want. Only as they gain experience with a developing design do they discover what they want and become able to define the requirements.

Many methodologies, and many more software projects, have foundered on this fact that requirements cannot be fully known at the outset. With painful regularity, traditional methodologies have produced, at the cost of hundreds of thousands of dollars and many man-years of effort, fully functional, complete systems which are unusable because they do not do what the customer wants them to do.

Another difficulty with traditional methodologies is the loss of meaning and understanding that frequently occurs in the translation from analysis to design and from design to implementation. Often three different representations are involved:

different kinds of drawings for analysis and design, and code for the implementation. The challenge of maintaining consistency and understandability between them is called "bridging the semantic gap"; frequently the gap is not successfully bridged.

Still another problem with traditional methodologies is that in focusing on getting the product completed correctly, they have failed to take adequate account of the inevitability of maintenance.<sup>25</sup> There appears to have been, in earlier days, a naive, unexamined belief on the part of many that a software system could be finished, made right, fully suited to the users' purposes. Once this was done, it was thought, the job was finished. Gradually software developers have become aware that no system is ever finished, unless it is no longer being used. Many have found, to their dismay, that up to 80% of their software development costs come in fixing and adapting their product after delivery. Developers are always aiming at a moving target, because users' purposes and the computational environment are always changing. (A fundamental element of this change, which still seems to be poorly understood, is that in using the system, people learn better what can be done and what they would like; ipso facto their purposes change.) Change is inherent in the software world (as it is in the rest of the world, of course.)

New methodologies are being developed which come to grips with the lack of clear requirements, the tacit, dispersed nature of knowledge, the importance of semantic consistency among analysis, design, and implementation, and the inevitability of

---

<sup>25</sup> Again, software maintenance, as the term is generally used, is not a matter of preventing physical deterioration, but of fixing bugs as they appear, and maintaining complementarity with the surrounding environment.

maintenance. Some of these take advantage of object-oriented technologies, which we introduce below. These approaches generally involve some form of prototyping in the early stages. Prototypes are used as a vehicle through which the designers and users of the new software can come to understand their respective capabilities and needs, thereby establishing system requirements. Because object-oriented programming environments are flexible and pre-supplied with components that can be tailored to new purposes, they enable rapid prototyping, in which a prototype can be quickly evolved through several iterations in a kind of dialogue between designers and users.

Object-oriented technologies are also designed to bridge the semantic gap by allowing the entire development process, from analysis through coding, to use the same terminology. Those who will use the system as well as the designers and the programmers who do the nitty-gritty implementation may think about the problem being addressed in terms of the elements of the system and their interactions: these are represented in the evolving software as objects and their methods. Instead of having to translate from design diagrams in one notation to code in another, object-oriented programmers doing detailed implementation fill in the details of the interactions of the objects developed in analysis and design. A related advantage of using the same kind of notation throughout is that analysis, design, and implementation can all be occurring simultaneously (as is often necessary as requirements evolve).

Proponents of object-oriented techniques claim that they also improve the maintainability of software systems, because their modular structure is understandable, and allows changes to be localized. Object-oriented systems



generally avoid the problems of "spaghetti code" in which one small change made here necessitates corresponding changes all over the system.

### 4.3. Tools

It is difficult to discuss methodologies without considering development tools at the same time, because the two are highly complementary, and often designed to be so. Of course there has been extensive evolution of programmers tools, aimed at helping with virtually all the aspects of software development. Among these are of course programming languages, which we have mentioned. Also there is an increasing number of programming environments, which provide a suite of tools in addition to the language proper. Some of these tools include

- **Debuggers** – these help programmers find and fix mistakes on the screen. (In the early days, one had to get a printout of the program and look through the code by hand to find the error.)
- **Compilers** – these translate the more abstract code written in higher-level languages into machine code (binary or executable code) that the computer can run. Good compilers are remarkable in their ability to make tradeoffs leading to efficient use of machine resources.
- **Diagramming tools** – these are an important kind of CASE tool. They automate the process of drawing the extensive diagrams often used in traditional analysis and design. While they are not much faster than drawing by hand initially, they have the advantage of speeding up (the inevitable) changes considerably.

- **Code generators** – these translate from a higher level specification of some kind, for example certain highly structured kinds of design diagrams or screen layouts, to code. They are especially useful for creating the code necessary for creating user interfaces and reports. More capable and accurate code generators is one of the most sought after, and elusive, goals of CASE.
- **Version control tools** – these have been developed in response to the challenge of coordinating the work of large teams of programmers. They keep track of the different versions of different modules, facilitating team development, and helping integrate changes. For example, if module A is used by modules B and C, but then module A must be changed for some reason, a good version control tool will alert programmers to the dependencies so that they can adjust B and C, if necessary, to restore compatibility.
- **Browsers** – these are relevant primarily to object-oriented languages, which make use of structured hierarchies of abstract data structures called classes. Class hierarchy browsers allow programmers to look through the hierarchy easily, browsing for classes that may be useful to them. In general, browsers allow programmers to examine different aspects of programs and systems from a variety of different viewpoints. These different viewpoints give them a better grasp of different aspects of the complex systems they are building.

#### 4.4. "Automatic Programming" and augmentation of human creativity

To what extent can the process of software development be automated? Computers can do so much, can they produce software? How necessary are people to the software production process? These questions concern the potential of automatic

programming and the more general subject of automated support for software engineering. Some have held that software production can be automated, and point to developments which they claim to prove their case. There is a fair amount of attention given today to automatic code generators, which automatically produce executable code from diagrams or other visual representations of program concepts. Some CASE tool builders provide this kind of capability, at least in limited fashion.

Another school of thought holds that automatic programming is a chimera, that only people write programs, and that the idea of automatic programming is fundamentally mistaken. There is a great deal that computers cannot do in producing software; they can do none of the interesting, hard problems.

These positions, though ostensibly in conflict, are reconcilable when couched in a different way. As we will see in the next chapter, the meaning of automatic programming has evolved in a revealing way. For now it suffices to say that while some kinds of activities can be automated, others appear to be impossible to automate. But unquestionably computer tools can help people in their tasks, by augmenting human capabilities. (Englebart 1963)

The dispute about the potential for automatic programming, and its de facto resolution in the nature of the new tools and processes being developed, point up an important aspect of the evolution of programming practice. That is, software engineers are gradually discovering and accepting that software development is an on-going process and must be treated as such. In general terms familiar to economists, the capital structure is not static; therefore capital goods, to maintain their value – their position of usefulness in the evolving capital structure – must evolve. Because the software industry has learned that change is inevitable, both in

the initial development period and after products are put to use, many of the most useful languages, tools and methodologies now being developed are those that help software developers manage change. Of these, perhaps the most important are the object-oriented technologies. We finish this chapter with a short introduction of these.

## 5. Object-Oriented Technologies

Object-oriented programming systems (generally known by the disarming acronym OOPS) have their origins in the programming language Simula, which was designed to enable the construction of computer simulations. The units of interest in Simula are the objects in the system being simulated. What made Simula different from previous languages is that the modules from which its programs were built were composed not of functional units only, like traditional subroutines, but combinations of functions and related data. From this idea, object-oriented technologies were borne.

An object, then, is a bundle of data and related functionality. The concept is natural one, applicable to modeling natural systems. Consider an airplane, for example. This is an object defined by certain data, including its cruising speed, carrying capacity, age, location, etc. as well as by the functions that it can carry out, such as taking off, cruising, landing, and taxiing. In the pure object-oriented systems such as Smalltalk and Eiffel,<sup>26</sup> everything in the system is an object; the approach is

---

<sup>26</sup> There are also hybrid systems such as the popular C++ , which has some of the features of object-orientation and lacks others. Most of this discussion pertains to

consistently applied. Let us consider some of the key concepts of object-oriented programming.

### 5.1. Encapsulation

One of the most important characteristics of object-oriented programming systems is that they achieve a higher degree of modularity than previous styles of programming.<sup>27</sup> This is because of what is known as the encapsulation of data and function. Recall that in older languages, while a certain degree of modularity is possible through the use of subroutines, there is still a significant chance for interference between modules because these modules generally share a common pool of data. The result is programming's version of the tragedy of the commons: one module often changes the data or its format in such a way as to confuse or make meaningless another module's use of that same data.

In object-oriented languages, by contrast, each object's data is encapsulated along with its own methods, and care is taken not to allow other objects to interfere with that data. Consider a possible software object *airplane* perhaps representing a real airplane in a navigation system. Its data might include *airspeed*, and *heading*; its methods might include *accelerate*, *decelerate*, *turn\_right*, and *turn-left*. In a properly encapsulated object-oriented system, only the *airplane* object itself has

---

the pure object-oriented languages, and especially Smalltalk, with which I am most familiar.

<sup>27</sup> At least they can achieve it. It is perfectly possible to write spaghetti code in an object-oriented language, just as it is possible to write elegantly modular code in a traditional language. It is simply harder in each case.

access to its *airspeed* and *heading* data, and they can be changed only by the *airplane*'s invoking one of its methods. It is not possible for some other object in the system to change that data, by accident or error, as would be the case, say, if the *airspeed* and *locations* of all the *airplanes* in the system were stored in a common, generally accessible table.

Object-oriented languages thus provide programming some of the benefits that property rights provide economies. Indeed, Mark Miller and Eric Drexler hold that the development of object-oriented programming constitutes an independent rediscovery by programmers of the virtues of property rights.<sup>28</sup> Just as property rights secure to economic agents a sphere of autonomy, and a confidence that the possessions for which they make plans will not be interfered with arbitrarily from the outside, encapsulation provides software objects an autonomy and security that the data they depend on will not be interfered with. The upshot is similar in both settings: just as property rights foster coordination in the economy, encapsulation fosters coordination in software systems.

## 5.2. Message Passing

In object-oriented programming, this encapsulation is supported by a means of inter-module communication called message passing. Objects do not directly change other objects; they "send them messages" to one another requesting services. Each method (be alert here to distinguish the distinct, but closely related concepts of method and message) that an object "knows how" to carry out can be

---

<sup>28</sup> See Miller and Drexler (1988) for a provocative discussion of this idea.

triggered by that object's receiving a corresponding message. If the object "understands" the message – that is, if it has a corresponding method in its repertoire of functionality – it performs that method. (If not, it triggers an error message in the system and the programmer gets to do some debugging.) In no other way can one object in a system interact with another. Again, this expedient serves an important security function: one part of a program simply cannot interfere with data encapsulated in another. It has no means for doing so. All it can do is send an appropriate message asking for, say, some part of that data or that an operation be performed on it.

### 5.3. Polymorphism

One of object-oriented technology's most powerful means of helping programmers manage complexity is polymorphism. This is a daunting name for an entirely familiar concept from everyday life. Polymorphism is the assignment of the same name to different but related actions (methods, in Smalltalk terminology.) If, for example, I were to ask you to shut the window, and then ask you to shut the door, you would not be confused. You would interpret "shut" in two different, though related ways appropriate to the two different contexts. Windows are shut with a different set of actions that doors are shut. In like manner, object-oriented programming languages interpret the same method name in different manners appropriate to the context, that is, appropriate to what kind of object is involved. That is polymorphism.

As simple as it sounds, it has been tremendously empowering to programmers. No longer do they need to compose different names for each slightly different action in slightly different context. They use the same appropriate term in all contexts, but

implement the methods appropriately differently for each kind of object. With polymorphism, programmers can address the great complexity of dealing with many slightly different kinds of objects by the simplifying power of abstraction, as we do in everyday life with our various different meanings for "shut" (shut off the TV; shut the book; shut up). Polymorphism lets us avoid addressing complexity with complicatedness, as in programming languages which would require a different kind of "shut" for each context (e.g. `shut_window`, `shut_door`, `shut_tv`, `shut_mouth...`).

#### 5.4. Information Hiding

Polymorphism and message passing make possible information hiding, another important characteristic of object-oriented technology that serves to simplify the programmer's job. Information hiding has not to do with secrecy, as it might sound, but with fostering the division of knowledge by making it unnecessary for programmers or objects to have much information about the other objects with which they interact. The key point is that all a programmer or object needs to know about another object is what useful services it can provide, and what messages it must be sent to trigger those services. It is not necessary to know anything about how the object actually does what it does.

The analog to everyday life is strong. When we deal with an accountant, for example, we might ask her (send her the message) to figure out our tax liability on a certain transaction. All we need to know is what message to send her to get her to perform the desired service. We do not know, nor do we want to know, exactly how she does it. That would defeat the whole purpose of the division of labor. It



would distract us with unnecessary knowledge, and might lead us to give the accountant unwanted advice as to how to do her job.

There is another benefit. That is the interchangeability of implementation. When a programmer works out an improved method for some kind of object, he can simply pull out the old implementation and put in the new. As long as the message that triggers it remains the same, no one else need know, and no other kinds of objects need be changed; changes in one module necessitate corresponding changes in other modules far less frequently in object-oriented programming than in conventional programming. Hence information hiding is an important enabler of software evolvability.

#### 5.5. Classes and inheritance

In object-oriented programming, every particular object in any actual program or system is an object of a particular kind or class. As such, it is called an instance of that class. Classes themselves are an important kind of abstraction, called abstract data type. There can be thousands of instances of a certain class, or none. Class is the abstraction, the kind of object. Here is another valuable abstraction mechanism. With the help of classes, which abstract from the particular characteristics of particular objects to comprehend what those kinds of objects all share, programmers have another means of getting a grip on complexity.

Furthermore, these classes are organized in inheritance hierarchies, which help make clear what kinds of things they are, and allow the sharing of characteristics.

David Taylor explains classes and inheritance as follows:

Inheritance is a mechanism whereby one class of objects can be defined as a special case of a more general class, automatically including the methods ... of the general class. Special cases of a class are known as subclasses of that class; the more general class, in turn, is known as the superclass of its special cases. In addition to the methods ... they inherit, subclasses may define their own methods and ... may override any of the inherited characteristics. (1990, p. 22)

For example, we might have the general class of objects *vehicle*, with subclass *fourWheeledVehicle*, which in turn has subclasses *car* and *truck*. Classes *car* and *truck* would inherit all the methods of *fourWheeledVehicle* and *vehicle*, but each could specialize any of these methods as appropriate, and add additional methods as needed.

In David Taylor's words,

The invention of the class hierarchy is the true genius of object-oriented technology. Human knowledge is structured in just this manner, relying on generic concepts and their refinement into increasingly specialized cases. Object-oriented technology uses the same conceptual mechanisms we employ in everyday life to build complex yet understandable software systems. (1990, p. 24)

## 6. Summary

Dramatic improvements in computer hardware have relaxed the resource constraints that shaped programming practice in the early days. Relatively freed from resource constraints and increasingly ambitious in undertaking large, complex problems, software developers found themselves confronting a daunting complexity constraint – how to manage the complexity of the systems they were trying to build. In response to this challenge, a variety of tools and development methodologies have evolved to enable better abstraction capability, more modularity of system

design, and better conceptual grasp of the evolving systems. The recent development of object-oriented technology has provided substantial advances in programmers ability to manage complexity with effective modularity and abstraction.

## Chapter 3

### Designing new capital: lessons from software development

*Oh, I see the crescent promise of my spirit hath not set.  
Ancient founts of inspiration well through all my fancy yet.  
- Tennyson, "Locksley Hall"*

*In speaking with each other we constantly pass over into the  
thought world of the other person; we engage him, and he engages  
us. So we adapt ourselves to each other in a preliminary way until  
the game of giving and taking – the real dialogue – begins.  
- Hans-Georg Gadamer<sup>29</sup>*

#### 1. Introduction

In this chapter we look at the process of new software development, its problems, practices, and historical developments, to see what it may teach us about the nature of new capital development.<sup>30</sup> We will see new software development to be a social learning process, and identify important aspects of that process. In an effort to get a good grasp on the software development process as a whole, we will take

---

<sup>29</sup> (1975, p. 57)

<sup>30</sup> We need to keep in mind that virtually all capital goods are used jointly. Hence we will try to think about individual capital goods in terms of the contexts in which they are used, and think of software applications as systems of sub-programs that interact extensively with one another. Most software constitutes not so much a single tool as a system of tools (consider a word-processor, for example: it has many modules including its text editor, printer drivers, spelling checker, search and replace facilities, etc.).

two different perspectives on it. We focus first on its social aspect – its necessarily interpersonal nature – through an examination of evolving software development practice. Then we focus on its being a learning process – one in which knowledge grows, becomes coherent and embodied in a usable form – through examining the evolving high-order goods used in the process – the tools software engineers have developed to help them in their work. In the next chapter we will go on to consider the challenge of software maintenance, focusing on software development's being an on-going, never-completed process that occurs through time. Inevitably we will cover some of the same ground more than once from different angles. I hold this to be a strength of the method rather than a drawback, however, because the software development process, like a software system itself, is a complex system beyond our complete understanding, but which we can understand better and better by taking a variety of different views into it.

## **2. Discovery in the design process: why prototyping**

Software development, like all capital goods development, is a social, not a solitary process. Many people are involved, because many people must contribute their own special knowledge to the evolving system. The nature of this social interaction, through which many people's different knowledge is brought together and embodied in new capital goods, is not straightforward. In particular, it is not a matter of saying to each, "Tell me what you know that's relevant," and incorporating that in a straightforward translation of some kind. Frequently we do not know what knowledge is relevant, nor could we express it clearly if we did.

And yet much traditional software methodology has proceeded on the assumption that we do and can.

An alternative approach which is growing up and gaining acceptance is the use of various techniques aiming to discover what knowledge is relevant – what needs and opportunities this tool may address – and to express that in usable form: embodied in the evolving design of the new system. 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 in the new capital.<sup>31</sup> The medium for the dialogue is the prototype itself – the emerging design. In a useful sense it is the prototype, the new design, 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.

### 2.1. Divided knowledge in software development

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

---

<sup>31</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.

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 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, like much knowledge in general, is tacit, inarticulate. (Polanyi 1964) Accordingly the most fundamental kind of knowledge necessary to the tool-building process – what the tool 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 tool builders to find out? Prototyping provides a means.

Prototyping is an iterative process that accommodates adjustment and change; it anticipates instability of requirements:

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 more frequent, regular interaction. Because the prototype is a version of the evolving tool, 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.

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,

Software 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)

Among certain members of the mainstream CASE community, the significant and on-going challenge of establishing what software systems are to do is now being recognized. In "A Self-Assessment by the Software Engineering Community," summarizing the findings of the International Workshop on Computer-Aided Software Engineering, Gene Forte and Ronald Norman write that "Prevention [of defects] begins with better ways to capture, represent, and validate the objectives and requirements of systems we are trying to build..." "There is still much work to be done in defining generic [software development] processes..," they say. "Areas that are particularly weak in process definition [include] requirements elicitation..." (1992, p. 29)

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.
- informed with this new knowledge, the developer improves and extends the prototype, and offers this new 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.

We emphasize that 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. But developers need to avoid naive prototyping. If an actual product is actually to be shipped on a reasonable budget, developers must avoid what is known as the "creeping feature" problem, in which more and more functionality is always being planned in, so that the product is never finished. On the other extreme, developers must resist the temptation to ship the prototype.

Sometimes clients are so please with a prototype, that they say, "We'll take it," before important issues of robustness and efficient implementation are addressed.

Sometimes the product to be delivered is implemented in a language different from the prototyping language. In such cases there is more of a difference between prototyping and implementation than in cases where the delivered product is a fully worked out descendant of the last prototype. Even in these latter cases, however, in which there can be an almost seamless transition from prototype to implementation, it is important for the developers to distinguish clearly between the different goals of prototyping and implementation. Otherwise they can fall prey to feature creep, slipped deadlines, and broken budgets.

The growth of prototyping in software development is a tacit recognition in the software industry that knowledge is more tacit and more dispersed than has previously been recognized. The shift from the traditional, "waterfall" type methodologies to methodologies that depend on prototyping seems to represent a shift in view of software development. It is coming to be seen less and less as a matter of manipulating static knowledge and more and more a matter of dynamic learning.

Capital development appears to be fundamentally a matter of dynamic learning. Because the knowledge that must be embodied in new capital goods is constantly being developed, widely dispersed, incomplete, and frequently tacit, a learning process which elicits and brings together this knowledge is essential. The point applies not just to software, but to capital goods in general. In manufacturing

industries in particular, there has been recent work emphasizing the importance of the "prototype/test cycle" and effective team learning.<sup>32</sup>

## 2.2. Knowledge of function, knowledge of design, knowledge of implementation

At this point let us step back and look at the software development process from a broad perspective. There seem to be three general stages to the process, which correspond directly to the broad categories of knowledge about tools discussed in Chapter 1: knowledge of function, of design, and of implementation.<sup>33</sup>

1. Establishing the requirements. What is the software to do? What is this tool supposed to be able to accomplish? This knowledge of function comes primarily from the tool user.
2. Design. What sort of tool may be fashioned so as to provide the desired functionality? What sort of design would best meet the users' requirements? This knowledge of design comes primarily from the designer, the specialist tool-maker.
3. Implementation. This is the actual instantiation of the design, the coding process. How, precisely, is this design realized? How may the details of

---

<sup>32</sup> See in particular Womack et. al. (1990) and Wheelwright and Clark (1992). We will take up the application of these insights to physical capital goods in Chapter 6.

<sup>33</sup> These cannot be sharply partitioned, either in time or in the nature of the development activity. "Final implementation," for instance, nearly always involves elements of design, as the programmer figures out the best way to implement a particular algorithm; and "design" encompasses many high-level implementation decisions.

construction be arranged so as to achieve good performance in speed and efficient use of machine resources? This knowledge of implementation comes primarily from the skilled programmer.

Prototyping is a valuable part of software development because these kinds of knowledge are dispersed and latent. They reside in different individuals who may not know one another and may have trouble communicating. Furthermore, the individuals may not be consciously aware of their knowledge, which needs to be brought out in application to the problem at hand. 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 the tool needs to do. Furthermore, the prototype itself provides the medium in which these different kinds of knowledge may be captured.

Knowledge of implementation, finally, often resides in still others, the actual coders. Once the clients accept the prototype as offering what they need, the designer often turns over the implementation job to programmers who specialize in efficiency of implementation: they construct the design to run with the best possible balance of high speed and low memory use on the computers for which it is intended.<sup>34</sup>

---

<sup>34</sup> As we shall see, the kinds of knowledge involved in design and in implementation are not independent, because designers must know what it is possible to implement, and implementors essentially design the details of their implementation. Nevertheless, these two types of knowledge are conceptually distinct and may not be concentrated in the same person.

The nature of prototyping brings out not only that the necessary knowledge is dispersed, but also that it is frequently tacit. Prototyping advocates stress that clients cannot say what they want. Even when they seem to know at some level, often they cannot express it. The prototyper must discover this by showing them different capabilities and carefully attending to their responses. 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. Indeed, the various books on prototyping are attempts to make more explicit some of the tacit knowledge that software designers themselves have developed over the years.

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 grows the knowledge necessary to building a software tool.

### 2.3. Software development as interactive learning

In short, the software development process exemplifies the classic Hayekian knowledge problem: the different kinds of knowledge to be coordinated are dispersed, tacit, incomplete. This being the case, the development of new software capital is a discovery procedure, an interactive learning process. Through this process, the dispersed knowledge is brought together in the new software tool. The knowledge gets built into, coalesces in, becomes embodied in, the software.

Thus in an important sense, it is actually the tool itself – the new software – 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>35</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 formal methodology. Traditional approaches to software development seem to make the same kind of assumption that is made in many neo-classical economic models: that all the relevant knowledge is available, and that therefore what remains is mechanically to work out its consequences, optimizing within given constraints. In

---

<sup>35</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)



this error we can see both why the traditional approaches to software engineering have led to cost overruns and frustrations for clients and developers, and why neo-classical economics is inadequate for illuminating the software development process. 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 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 rapid 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.

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 tool

Note that there is another kind of interaction at work here: that between the client and the tool 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 tool 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.

There would seem to be two aspects of this discovery process at work 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 user's discovering in the conscious, articulate part of his mind, the knowledge that was always there in some sense inarticulately, 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. Tacit knowledge made more explicit through interaction with the prototype need not be made fully articulate (that is not possible), but only clear enough so that it can be communicated to the designer for incorporation into the next version.

Another, more subtle aspect of this discovery process has to do with the incompleteness of knowledge. Above we examined the bringing to light of knowledge which already existed, but not in communicable form. In some sense the users of the prototype knew all along that they wanted and needed certain capabilities in the software, but they were 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 occurred 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.

The tacitness and incompleteness of the user's knowledge of what they need are the main reasons for the failure of traditional methodologies in modern software development. Software requirements cannot be completely articulated in the first stages of development because the necessary knowledge is incomplete and because

much of it is inarticulate. Only through interacting with the developing tool do users discover and communicate to the designers what they need.

### **Interaction between designer and tool**

It is not only the users who interact with the tool, 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 the new knowledge which constitutes the evolving design. They are learning also. 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>36</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. In Mullin's description, software design is fundamentally a matter of learning:

The best way to do OO program design is to realize that you are dealing with systems, and the best source of information is the system you are

---

<sup>36</sup> Personal interview conducted October 1992. "Kent" is Kent Beck, another pre-eminent Smalltalk programmer.

duplicating or enhancing. Your job is not to dictate how the system will work, but to understand how the system already works. As you do this, you are acquiring valuable information about the classes you will need to construct your system and how instances of these classes interact with each other at runtime. (p. 36)

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 problem over time, and gradually puts the design together. In this respect software design would seem to be 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.

### **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:

In RP 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)

### **3. Designing as understanding: the role of tools for thought**

We concentrated in the previous section on the social, interpersonal aspects of software development. In this section we concentrate on the learning aspects. We do this through an examination of the higher-order tools software designers use to help them do their work of creating still other software tools. The fundamental challenge in software development is to make sense of the complexity of the systems we are trying to build: to understand them and the way they function, and to express that understanding in code. Most of the tools software designers use,



higher-level programming languages in particular, are tools for helping them understand what they are doing.

An important and illuminating exception to this rule is automatic code generators, both those that generate higher-level code from diagrams, and compilers, which translate higher-level code into machine language. Paradoxically, although these tools in a sense produce software automatically, without any human participation, a look at their evolution, what they do, and what they do not do, reinforces the fundamental point that software development is a wholly human learning process.

### 3.1. CASE tools

Let us start with a quick look at CASE tools proper. The mindset of mainstream CASE methodology is illustrated in the following passage from CASE is Software Automation (1989), by Carma McClure, an expert on computer-aided software engineering. Having defined CASE as "the automation of software development," McClure expands as follows:

CASE proposes a new approach to the software life cycle concept, that is based on automation. The basic idea behind CASE is to provide a set of well-integrated, labor-saving tools linking and automating all phases of the software life cycle....

Traditional software technologies are of two types: tools and methodologies. ... Most software tools are stand-alone, mainframe-based, and concentrate on the implementation part of the software life cycle.

The software methodology category includes manual software development methodologies such as structured analysis, structured design, and structured programming. These methodologies define a step-by-step disciplined process for developing software.

The CASE technology is a *combination* of software tools and methodologies. Furthermore, CASE is different from earlier software technologies because it focuses on the entire software productivity problem, not just on implementation solutions. Spanning all phases of the software life cycle, CASE is the most complete software technology yet. CASE attack software productivity problems at both ends of the life cycle by automating many analysis and design tasks, as well as program implementation and maintenance tasks.

Because manual structured methodologies are too tedious and labor intensive, in practice they are seldom followed to the most detailed level. CASE makes manual structured methodologies practical to use by automating the drawing of structured diagrams and the generation of system documentation. (1989, pp. 5-6, emphasis added)

There are a number of points here worthy of note. One is the acceptance of the idea of the traditional "software life cycle," which begins with analysis of the problem domain, and proceeds sequentially in a "a step-by-step disciplined process" through design and implementation stages. There are two important hidden assumptions here. The first is that the problem is known and awaits our analysis. The second is that implementation – actually writing code to solve the problem – properly occurs after analysis and design are completed. In this regard "the generation of system documentation" is also important. Traditional methodologies depend on extensive documentation of requirements and specification which are supposedly to be completed before coding begins.

Also noteworthy is the comment that "in practice [the structured methodologies] are seldom followed to the most detailed level." This is undoubtedly due in some measure to the tedium and labor-intensity to which McClure calls attention, but it is probably due also to the awareness of those doing the work that because requirements and their corresponding specifications are never really finalized, by

the time they could actually complete a structured design, the requirements would have changed and they would lose their labor.

Finally, note the implied connection between CASE and more traditional programming languages, especially structured programming languages. As we will see below, object-oriented technologies appear to offer a significant improvement over traditional CASE.

A survey of the main features offered in current CASE tools<sup>37</sup> reveals the following ten basic functions, which I have 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

---

<sup>37</sup> This particular listing is drawn from Kara (1992).

With the exception of the last category (which we take up below in the section on automatic programming), each kind of tool is devoted, in its own way, to helping the 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 example, sometimes designers will indicate all the inputs necessary to a particular module, but fail to specify any output. Diagram consistency checkers point out such flaws automatically.

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 experts. In this context it is very helpful to have a shared database where a variety of information about the project can be stored and accessed. Because the requirements of a system gradually develop and change as the system takes shape, it

is often helpful to have a history of their evolution, so that team members may understand why 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 in aiding learning we discussed in the previous section and need not repeat at length here: 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 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.)

### 3.2. Object-oriented programming environments

Programming environments such as Smalltalk provide some additional tools not included in the above list of standard CASE functionality. (Smalltalk and similar environments are not generally called CASE tools, even though they are certainly instances of computer assisted software engineering in the simple meaning of the term). The nature of these tools also points to the learning aspects of software development, and suggests why Smalltalk has become popular for prototyping.

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>38</sup> The technical term for this is incremental compiling. The 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 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 this kind 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;

---

<sup>38</sup> Ward Cunningham, personal interview, October 1992.

you don't leave the context."<sup>39</sup> Additionally, incremental compiling leads to higher quality, because smaller chunks of code are easier to test.

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 also provides rapid feedback and hence clearer understanding. 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>40</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. Accordingly, the programmer then has to search for the problem. In so doing, she loses the context; her thought processes get interrupted. Additionally, the whole program usually has to be recompiled and rerun before she can make sure that she 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 her.

---

<sup>39</sup> Lee Griffin of IBM, personal interview, October 1992.

<sup>40</sup> Ward Cunningham, personal interview, October 1992.

The combination in Smalltalk of incremental compiling and the build-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 program in its newly repaired state, by pushing (with the mouse) the "Restart" button on the debug window. Smalltalk users find this feature extremely important.<sup>41</sup>

Another important feedback capability of Smalltalk is known as type checking. Smalltalk routinely checks the kinds or types of data that are being processed, making sure each data item is of a type which the method operating on it is equipped to handle. When Smalltalk discovers that this is not the case, it informs the programmer with a debug window showing where the incompatibility occurs. In a sense, then, Smalltalk looks for problems, on the assumption that problems will occur.

Of course problems – bugs – do occur in all programming, but most programming languages are ill-equipped to help developers deal with them. Indeed, many languages pointedly lack these type-checking facilities, because they slow down the execution of the program. 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

---

<sup>41</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.



program. Having type-checking going on is pointless, on this assumption, because the delivered program, by assumption, will be accurate; all the checks will turn up false.

Smalltalk, by contrast, recognizes in its very design that we live in a world of error. The designers of Smalltalk took very seriously the learning challenges of designing software, and therefore provided this type-checking facility to support software developers.<sup>42</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++<sup>43</sup> is that Smalltalk talks sooner and louder when you are doing a bad job."<sup>44</sup>

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 provides windows which display relationships between modules (objects) such as which kinds of objects send messages to others (messages

---

<sup>42</sup> I am indebted Ward Cunningham for explaining this distinction.

<sup>43</sup> Recall that C++ is a hybrid language with certain object-oriented features.

<sup>44</sup> Paul Ambrose, personal telephone conversation.

trigger actions by the objects which receive them). 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. In fact, the very value of the Smalltalk browsers and windows has stimulated the development of still other kinds of tools which give different perspectives into software systems, to make more understandable various different kinds of relationships. A complex system, by its nature, cannot be wholly understood. But it can be understood better and better in proportion that one has a variety of different perspectives on it. Each new perspective enriches one's understanding of the other perspectives, and hence of the system as a whole.

The common characteristic of these programming tools is that they all serve to aid the programmer in understanding the evolving software system. Such tools are valuable because software development is a learning process; by their nature they suggest how knowledge-intensive software development is. These tools help programmers learn what they are working with and what remains to be done. They show us that software development is not a mechanical matter of translating product requirements to code, but of learning what the software is and needs to become.

On this point, Mark S. Miller, formerly of Xerox Palo Alto Research Center and now co-architect of the Xanadu hypertext system, 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>45</sup>

Let us turn now to tools that generate code from diagrams, and other tools that, in general, relieve the programmer of writing code. In their evolution are more interesting lessons about software development as a social learning process, lessons which reemphasize the fundamental point that the development of new capital goods – whether software or more physical capital – is not a mechanistic affair, but a creative, dynamic learning process.

### 3.3. Automatic programming

Automatic programming is a term we hear rarely now, but it refers to an important dream of the software community. Among its descendants is the automatic code generation provided by certain CASE tools. The goal of the advocates of automatic programming was to have computers, rather than people, write programs. As Mark S. Miller explains, there are opposite opinions of its success, and each opinion, viewed from its own perspective, has validity.<sup>46</sup>

One view is that automatic programming was a total failure. Look around us; there are millions of people writing programs, in a process that is anything but automatic.

---

<sup>45</sup> Mark S. Miller, personal telephone conversation.

<sup>46</sup> I am indebted to Mark S. Miller as the source of most of the insights of this section and the next. Quotations are from my transcription of a telephone interview with him unless otherwise noted.

Computers can't write programs; programming requires human imagination and creativity. On its terms, this majority view is certainly valid.

But the activity we call programming today is a different activity from that which was called programming years ago, when automatic programming was first advocated. At that time, says Miller, programming

was largely low-level assembly hacking; for example, it was concerned about what operand was in what register of the machine. As far as that activity is concerned, the advocates of automatic programming succeeded. They succeeded in automating what programming was then.

This success is thanks to the development of compilers for higher level languages. As Miller says it, "we now specify what computation needs to happen, and the implementation in particular machine instructions is handled by compilers." We make this specification in higher level computer languages – languages which allow us to specify what is to happen in terms more abstract than the computer can handle directly. The compiler then transforms this more abstract coding into machine code that the computer can read. (Obviously each different computer language requires a different compiler for any given kind of machine.)

The historical change in terminology on which this disagreement turns is revealing about the very nature of software development. Miller explains:

[T]here was an incremental and gradual transformation over time of what it means to program. The transformation was from programming's being primarily implementation-oriented to its being specification-oriented. The implementation issues that were much of the programmer's concern in the old days are now handled by compilers.

To specify, in the software development context, is to state precisely what the program must do. In standard software engineering usage, specification occurs in a

language more abstract than a programming language, typically a natural language.

Miller continues,

However, there is an extraordinary number of levels of abstraction in the program. So when we think about specifying, at any point in the evolution of programming languages, what we mean is conceptual activity a few levels above where our programming languages are. There's too much grungy detail in the languages for us, so we specify with higher-level abstractions. What we do when we go from specification to the current level of abstractions that our languages allow us to operate at, is now called implementation. A few years ago, yes, that implementation would have been seen as specifying, because then we had no languages that could handle that level of abstraction. But the particular tasks our term implementation refers to change over time, with our capacities. At any time in the evolution of programming languages, we see the level of abstraction that our languages permit us as "too much grungy detail."

What we called specification yesterday – activity which is specification from the perspective of lower-level languages – we call implementation today, because today we have programming languages that allow us to capture and express that level of abstraction, with compilers to do the work of transforming that abstract expression into machine code. Over time, as programmers' essential higher-order tools of production – their programming languages – have improved, what it means to program has changed.

What can we learn about the development of software capital from this slice of the history of programming? There are at least three lessons relevant to our present purpose, learning how the capital structure expands and improves.

First, there is a necessary role for the human imagination in addressing the particulars of each new capital need, in this case, each programming challenge. Computers cannot figure out what must be done to solve a particular new kind of

challenge.<sup>47</sup> This kind of task is fundamentally a learning process – a matter of understanding the problem, and adequately expressing a software system that can address it. For the purpose of this expression, higher-level languages that enable us to express abstractions better are very helpful. We address this point at length in the next section.

Second, today's programming, i.e. design, is a profoundly social process in that it is entirely dependent on the division of knowledge embodied in tools. Everyone who uses a higher level language depends, for the realization of one's program in executable form, on the creativity, knowledge, and expertise of those who built the compiler one uses. Those who built the compilers have addressed for us, ahead of time, "the grungy details" of machine instructions. Their knowledge, their experience with what works well and what does not, is embodied for us in the compiler we no longer notice. Because they have taken care of lower-level automatable concerns, they free us to concentrate our efforts at higher conceptual levels.

In a similar category to that of compilers are the tools for generating code from diagrams or screen representations, and the tools for porting a system from one kind of computer to another. Examples include graphical user interface builders, code generators offered in certain CASE tools, and some visual programming languages. In each case they let one specify what is wanted in one medium, generally higher-level and more abstract, and turn that specification into code more accessible to the

---

<sup>47</sup> Perhaps profound advances in artificial intelligence will make this possible someday, but that day has not arrived.

machine (or, in the case of tools for porting between different computers, to the target machine). What all these devices have in common is that they embody knowledge of how to transform one representation into another. They perform the transformations for us, freeing us to concentrate on the substance of what we want transformed.

A third lesson taught by this history is that as capital goods improve, there is a concurrent, complementary development in what people using the tools know and do. This is a very important species of social learning: what the relevant community – in this case the programming community – does in their everyday work advances; the community learns. Over time, the programming community has built up knowledge of how to make efficient use of raw computer resources – how to manage the grungy details of machine instructions. It has also built up knowledge about what kinds of expressive capabilities are needed in computer languages. All this knowledge has been built into a set of gradually improving languages, and their related compilers. In an important sense, then, this community has learned a lot about programming. The whole community is in a sense smarter in their programming practices and tools. The change is reflected in the fact that what we mean by programming is completely different now from what it was twenty-five years ago. The change has been a social one in that the new knowledge is not to be found in particular individuals, but in the whole pattern of interaction among people, tools, and practices. Individuals don't necessarily know more – in many cases they are clearly able to be effective while knowing less than their predecessors – the knowledge that has developed is spread throughout the community, in tools, languages and practices over which no one individual has a complete grasp.

### 3.4. tools for aiding dialogue

There are many ways in which object-oriented languages try to address the problem of social learning in design, some of which we have discussed already, and others of which we will take up in the next chapter. In the present context – the development of new software – probably none is as important as this: the "pure" object-oriented languages such as Smalltalk 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 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, as we have discussed, 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, not because they are still more abstract than other recent languages, but because they let us create our own vocabulary, tailored to the problem space as we understand it, both for thinking about the problem and for implementing a solution to it. 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)

We have said that 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 problem domain, they avoid loss of meaning in translation. In a sense, they shorten the conceptual distance between the knowledge that goes into the new capital good and the good 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.

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 – learning about – complex systems. One important result is to facilitate communication among people with different kinds of knowledge to contribute to the software. Knowledge Systems Corp., a major development consultancy firm specializing in Smalltalk, has extended the object-oriented approach into a methodology (on which they are still working). They

have 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 methodology takes the idea of software development as a social learning process to its fullest extent. In order to begin developing the Smalltalk 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, and 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. We quote at length from Sam Adams' description of their experience:

While most methodologies rely on diagramming notations to attempt to capture and communicate complex interactions between objects,

roleplaying 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 roleplaying 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 roleplaying that can be only poorly captured on paper... An additional benefit of roleplaying 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 should note here that object-oriented technologies, and the methodologies aimed at rapid learning which they support, come at a cost; their benefits trade off against other considerations which make them inappropriate for some kinds of software projects. A language such as Smalltalk, which allows programmers to work at a high level of abstraction, generally runs more slowly than a language more oriented toward the concerns of the computer. In cases where speed of execution is paramount, it makes more sense to use a non-object-oriented language such as C, which makes optimal use of the machine's speed and memory. Also, where a problem domain is well understood, prototyping may add little to the programmers' understanding of what they must accomplish. Object-oriented languages and techniques are most valuable where exactly what is to be done is unclear, and where it is more important for the software do what is wanted than to do it as fast as possible.

Finally let us offer one somewhat philosophical perspective on software development as social learning. The accessibility of OOPS, the manner in which it empowers thinking about problems and expressing their solutions, demonstrates to what great extent learning occurs in the context of the social world, with its shared meanings captured in language. Higher level languages have increasingly let us

move away from the mundane concerns of the machine to concentrate on more general and meaningful abstractions. Software designers using higher level languages are much less distracted by the needs of the machine; their attention can be focused on the needs of the system they are building, in terms of the system and not the computer. Object-oriented languages and object-oriented methodologies such as that being developed by Knowledge Systems Corp. let us take a very large step in this direction, into the world of human discourse and imagination. In the objects and methods of object-oriented languages we have something akin to the nouns and verbs of the language of society. Accordingly, with object-oriented languages our powers of expression and understanding improve substantially, informed by the richness of meaning that comes with evolved language. Because software development is a social learning process, it gets easier as we become better able to do our thinking in terms of the social world we live in.

#### **4. Summary**

An examination of the tools and processes used in software development show it to be a social learning process. The process is a kind of dialogue in which dispersed, tacit, incomplete knowledge is brought together and embodied in new software tools. The process comprises interaction between users and designers, between users and the evolving tools, and between designers and the evolving tools. It is an iterative process in which the evolving tool itself serves as the medium of communication, with each new round of the dialogue eliciting more useful knowledge from the different people involved.

As programming practice has evolved, higher-order tools have been developed to facilitate the process. Some of these, such as compilers and code generators, serve to automate the clearly understood aspects of the process. These can be seen as freeing human effort to undertake, at ever higher levels of abstraction, the creative learning that is the essence of design. Most of the tools now used to facilitate the design process help software builders to get a better understanding of the complex systems they build. The most promising of these tools are the object-oriented technologies, which allow us to create the kinds of abstractions we need both to think about the problems effectively, and to specify their solutions.

## Chapter 4

### Capital Evolvability: Lessons from Software Maintenance

*Knowledge comes, but wisdom lingers, and I linger on the shore,  
And the individual withers, and the world is more and more.  
- Tennyson, "Locksley Hall"*

*...when you realize that much of the software problem has to do with building very complex systems that will run on networks with different kinds of hardware, and that no application will be considered done when shipped, you're inescapably led to a much more biological, modular system, for which something like objects will be required.*

*- Alan Kay<sup>48</sup>*

#### 1. Introduction

The process of software development does not end when the first version is shipped to the customer. It continues throughout the life of the product. The world changes, hence the software must change with it, if it is to maintain or increase its value as a useful capital good. Users' requirements change as their businesses change; the software needs new features to keep up with competitive products; it needs to run on new machines, to be used on networks, to drive new printers and plotters; etc. On the broadest view, as the economy grows and develops through the accumulation of new knowledge and its embodiment in new tools and new

---

<sup>48</sup> (1992, p. 13)

systems, software products must themselves "learn" – develop and improve – to maintain and improve their position of usefulness in complement to the other elements of the evolving capital structure.<sup>49</sup>

The process of adapting and enhancing existing software is known as software maintenance. It is challenging and costly. At present, the software industry is very concerned about maintenance, as evidenced by advertisements such as the following, which included a graphic of a hooded skeleton with a scythe, typing on a computer keyboard:

**Why Your Software Will Die Before Its Time.**

**Entropy.** It's the Grim Reaper of software development. As your code is modified and enhanced over time, its structure gradually breaks down. Until one day it simply can't be maintained anymore – not by you, not by anyone.<sup>50</sup>

The kinds of changes driving today's severe maintenance challenge, as well as their perceived importance, are suggested by the following lead copy from a twelve-page, four-color, glossy advertisement that was pasted into the November 2 issue of Computerworld, a major weekly news publication of the computer industry:

Today, information management professionals face more daunting problems than ever before. The applications you develop must meet business needs that seem to change daily. Mergers and acquisitions

---

<sup>49</sup> Lachmann writes,

...it is impossible to receive a permanent income stream unless its source has been kept intact, and ... this requires a problem-solving activity which may succeed or fail. Maintaining the value of capital resources is an important economic function. (1986, p. 73)

<sup>50</sup> Set Laboratories advertisement in CASE Trends, Vol. 4, no. 6, September 1992.

create demanding integration scenarios. The introduction of new technology brings with it the need for multiple platform deployment. You're feeling pressure for client/server processing from management and users alike. Meanwhile, the backlog of existing applications you need to maintain and enhance keeps growing.<sup>51</sup>

As computer systems have become larger, more complex, and more important to the success of enterprises, maintainability has assumed greater and greater importance. Software systems which are readily maintainable allow their enterprises to adapt quickly and smoothly to changes in their environment. Those systems which are not maintainable become a terrible burden, especially if they are essential systems. Accordingly, it has become more and more important to software engineers to build systems which not only work well now, but which also can be evolved without difficulty. In this chapter we examine, not the process of software maintenance, but the characteristics of maintainable software systems.

In the preceding chapter we described software development as a social learning process, and held that in an important sense it is the capital goods themselves that learn – 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, then, 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. Our task, therefore, is to look for the characteristics which allow software to embody new knowledge readily. These characteristics can be summed up in a single word: modularity. We will see that

---

<sup>51</sup> KnowledgeWare advertisement, insert, Computerworld, Vol. XXVI, no. 44, November 2, 1992.



because software development is a social learning process, modularity is essential to software evolvability. To continue the figure of speech of software "learning," in this chapter we will be investigating the aspects of modularity that allow software to learn.

Let us make clear here at the outset what we mean by software maintenance. The term may seem strange to those unaccustomed to its usage in the software field, because software does not wear out, and hence should need no maintenance. But as Hayek has stressed (1935), to maintain capital is fundamentally to maintain its value in the evolving capital structure of which it forms a part. Obsolescence is just as important as wear and tear. On this view, the term is not misapplied. It refers to any activities aimed at keeping software running as needed, from mundane fixing of bugs to adding enhancements.

As the term is used, however, it refers to more than simply activity which prevents software from losing value; it refers also to development of the software which may increase its value. As Sam Adams stresses, software "should be treated as a corporate asset that can appreciate through investment in its quality and reusability." (1992b, p. 6) In this work, by software maintenance, and related terms enhancement and evolution, we will mean any changes made to software aimed at maintaining or increasing its value by improving its usefulness in the evolving capital structure. We mean, in short, investment in existing software assets.

Note that we draw no sharp distinction between the activities involved in initial software development, and those involved in software maintenance. Indeed, many software developers mislead themselves in seeing these activities as somehow different and separate. Software development seems to be an ongoing learning

process, with much the same kinds of activities carried out whether a first version of a product has been shipped or not. The dialogue-like process that goes on among various users and designers at early prototyping stages continues in one way or another through "the maintenance stage." At this point, users are not reacting to a prototype, but rather to a delivered version of the product proper. Nevertheless, the users are still learning from the software, the designers (maintainers) are still learning from the users what is needed and from the developing software what is possible. There is continuity between initial software development and maintenance. The categorical distinction turns not so much on what the software developers but on the legal and contractual issue of whether an agreed-on first version has been shipped or not.

The reason we focus on maintenance for the purposes of this chapter is that in maintenance the greater or lesser ease of adaptation appears. By looking at software that is hard or easy to maintain, we gain insight into the design characteristics of evolvable software. But it should be noted that we are really interested in design issues – how do we initially design software so that it will be maintainable, so that it can be improved over time?

## **2. Evolvability as a design goal**

There is general agreement in the software industry that ease of maintenance is fundamentally important. Practitioners in the software world clearly expect continuous change, though they cannot know just what those changes will be. In Frank Knight's terms, they face uncertainty. (Knight 1971) They are foresighted,

though without clear vision of the future.<sup>52</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)

It appears that 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 have made the lesson clear. Change never ceases. Indeed, it seems to accelerate. Accordingly developers now try to build software so as to facilitate change in general. Good design, in an uncertain world, is design which prepares for change. A major goal of good software design, then, is to ensure design evolvability.

### 2.1. Co-evolutionary development

The evolution of complex systems, such as the capital structure, is not a movement toward some particular endpoint, or even in some particular direction. Evolution is

---

<sup>52</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)

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>53</sup> The development and availability of any particular technology changes the opportunity costs of developing any related good, whether substitute or complement, and thereby changes the appropriateness of any particular investment.

Consequently "the best solution" to a particular problem is a mirage that appears when one fixes 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 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. (Allen, 1990, p. 25)

Software developers, then, must try to build their products so that they can be evolved in such a way as to maintain a reasonably good fit in the evolving capital

---

<sup>53</sup> Other factors include people's expectations, the interest rate, availability of skilled personnel, etc. See Lachmann (1986) and Hayek (1935).

structure around them, regardless of how – out of a broad continuum of possibilities – that capital structure may evolve.

## 2.2. The Optimization Trap

Crucially, this means that optimization of software for any task as defined at a particular moment, should frequently 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,

...optimal efficiency would require perfect adaptation to a particular hardware and software environment, which is the opposite of portability,<sup>54</sup> and perfect adaptation to a particular specification, whereas extendibility and reusability<sup>55</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 eco-system: 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

---

<sup>54</sup> Portability is the ease with which a program built for use in one environment, e.g. on one kind of computer, can be adapted for use in a different environment.

<sup>55</sup> We discuss extendibility and reusability below.

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>56</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! (Allen 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 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 various workstations and PCs, communicating over a network spread across five cities. That species of software would be much more survivable were it less optimized and more evolvable.

### 2.3. 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:

---

<sup>56</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.

- **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. (Meyer, 1988, pp. 5-6.)

It is important to remember that all software, except for very simple, short programs, comprises systems of related functionality. To maintain awareness of the complexity of software, it is frequently helpful to think of it as being more like a factory, embodying a variety of machines and processes all working together, than like a single machine. From this perspective and in Lachmann's terms, software extendibility is a matter of capital recombination. In adapting software to changes in the specifications, some elements of its functionality are eliminated, some replaced, and others added; in much the same way that in retooling a factory to new production demands, some machines or processes are eliminated, replaced, or added. Software extendibility is the ease with which these changes can be made.

Similarly, software compatibility is matter of capital complementarity and (multiple) specificity. A software application is compatible with others when a complementary relationship can be easily established with them. It is incompatible when the different packages are so highly specific to some original purpose or context that they cannot easily be made to work together.

In this discussion it is important to remember that our attention here, as throughout this work, is primarily on designs, rather than on particular instances of designs. For instance, we are more concerned with how hard or easy it is for Microsoft Corp. to evolve the design of Word for Windows – enhancing it or enabling it to work

smoothly with some other programs – than with how hard or easy it is to change the copy I use to write these words. Similarly, in applying the lessons we learn here to "hard" tools, we are more concerned, say, with how easily a locomotive manufacturer may design the next generation of locomotive, than with how easily some railroad company may rebuild a particular engine to achieve higher levels of performance. Maintaining the value of particular instances of capital goods is important (especially when doing so in fact involves design changes), but our focus here is more on how the design itself – the state of the art in word processors or locomotives – evolves. This outlook seems consistent with Austrian capital theory. When Lachmann refers to a vintage locomotive's gradually being relegated less and less important duty, and ultimately to the scrap heap, he makes clear that it is "kicked downstairs" further and further by "the march of progress" – not by newer instances of the same model, but by a succession of newer designs using better technologies.<sup>57</sup>

The challenge of software maintenance, with its corresponding imperative that software be evolvable, casts an interesting light on the work of Hayek and Lachmann on capital maintenance and capital evolution. Both address issues of restructuring, of investments and capital combinations, when inevitable changes occur. Neither, however, emphasizes the issue raised here, of maintaining flexibility in the capital structure so as to be able to cope with future changes that cannot be fully anticipated. Lachmann, for example, in Capital and Its Structure speaks of "the changing pattern of resource use which the divergence of results

---

<sup>57</sup> (1978, p. 38). Lachmann quotes an elegant passage from Dynamic Equipment Policy (1949, McGraw-Hill), by George Terborgh.



actually experienced from what they had been expected to be, imposes on entrepreneurs." (1978, p. 35, emphasis added). Similarly, in "Another Look at the Theory of Capital" he says,

The capital stock in existence always contains 'fossils', items that will not be replaced and would not exist at all had their future fate been correctly foreseen at the date of their investment. (1986, p. 61, emphasis added)

Focusing as he does on changes that must be made in the capital structure when entrepreneurs incorrectly forecast the future, Lachmann may seem to suggest that entrepreneurs fully commit themselves to their vision of the future, tying their capital investments tightly to the future needs they anticipate, and allowing themselves no flexibility to adjust if events take a different path. In such cases we can properly speak, as Lachmann does, of "failure" and "error."

Here we suggest that frequently entrepreneurs do not commit themselves so completely to a particular view of the future, but rather make their best estimate of a range of likely outcomes, and build into their capital goods a flexibility with which to cope with this range of outcomes. There is a tradeoff here, of course. More flexibility will generally mean less perfect suitability to a particular set of circumstances, and some entrepreneurs might choose to bet their companies on the details of their foresight, seeking the higher return that will come from greater suitability. Others will accept a slightly lower prospective gain, building in more flexibility to allow them to adapt better. The upshot is very much the same, of course: there must be constant adjustment because the future was not, and could not be, correctly anticipated in all its detail. But many of the imperfectly adapted

capital goods in use at any time can be seen as imperfect not as a result of failure, but as a result of planned flexibility.

### **3. Evolvability through modularity**

It is generally accepted in software engineering that modularity is crucial to software extendibility, compatibility, and also reusability, which we take up below. Why? How does modularity facilitate 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 section.

#### 3.1. How modularity promotes evolvability

Simply stated, modularity leads to evolvability because in order for a software system to evolve smoothly, its overall structure must allow the maintainers (enhancers) of the system to pull out some part of the system's functionality and replace it with better, and/or to add new functionality, without too much difficulty. When software architecture is appropriately modular, with functionality encapsulated in relatively independent modules, these changes are relatively easy, because they are confined to a few modules. In non-modular architectures, by contrast, there are lots of interdependencies among different parts of the system which make the adaptation or extension very hard to accomplish, because so many different parts of the system are affected.

In this respect it is essential to note that the sheer amount of work involved is usually not the issue; the issue is grasping what work is to 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 bringing the whole system into coordination when a change is made. But more important than simply doing all this work is the danger that it will not be clear what must be done. A non-modular system will be significantly more difficult to understand than it might be. Accordingly, when functionality is added or changed, 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. Here again is the complexity constraint we mentioned in Chapter 2. Software development is a learning process; if a system cannot be understood, then further learning in respect to it is encumbered. 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.

Modularity makes possible the evolution of extremely complex systems because the 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 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.

### 3.2. Kinds of modularity

What, exactly, do we mean by modularity? What are its aspects? There are several, and they are not all complementary. 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 multiple rather than single specificity. 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. It is a matter of great importance in software development; we take it up below in section 5.

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, because if understanding one module requires knowledge of many others, it is difficult for someone to specialize.

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>58</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

---

<sup>58</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.

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 Post Office switched from five-digit zipcodes to the present nine-digit zipcode. Many software systems did not localize their treatment of zipcodes, 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 – problems that occur when the software actually operates, not problems that occur in getting it to operate. 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>59</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>60</sup>

#### **4. Design principles that yield modularity**

Now that we have examined the benefits and meaning 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 capital to evolve well. Putting it metaphorically, we are asking what makes software flexible.

Kent Beck, a well-known Smalltalk expert and president of First Class Software, 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>61</sup> (It is often necessary despite the problems we saw in the last chapter: that necessary knowledge is often unavailable until users

---

<sup>59</sup> Personal interview, October 1992.

<sup>60</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.

<sup>61</sup> Personal interview, October 1992.



have a chance to see and use a running version. The point is that when program development is done in an unforgiving programming language, there may be no alternative.) One of the main downstream costs 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 when a team perceives some kind of major change they would wish to make, they must proceed with their current, inferior design because there would be just too much to change to get the program the way they would like it.<sup>62</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, it becomes far more possible to let analysis, design, and implementation occur together, without encountering excessive downstream costs and problems. When the medium is flexible enough, it is not so costly to make changes downstream as one learns. In brief, maintenance is easier.

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. (Meyer 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 work on it, generally in groups. These principles facilitate that group effort.

---

<sup>62</sup> Lee Griffin of IBM Corp., personal conversation.

#### 4.1. 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 programming language 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 which 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 complex 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.

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 simply to remember what happens in the module obstructs programmers' progress; it is much easier to work with a statement such as, "`PurchaseOrder new initialize.`"

#### 4.2. 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 those to which it connects will have to be changed also. Thus, for the sake of continuity, the number of interconnections – interfaces, in software terminology – should be restricted. Restricting the number of interfaces helps maintain the division of knowledge, because those responsible for interacting modules must coordinate when changes are made, and if only a few modules interact, then there is less coordination overhead, 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 center, which are responsible for reconciling their interactions. 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.

#### 4.3. 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 perhaps 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.

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. This hurts evolvability, because when some part of that detail or data changes, the modules must be rewritten, and when errors occur, they propagate widely. Furthermore, when one module has access to

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 information in the design of their own modules. If and when that information changes, module design must change, too. Moreover, when modules communicate too much information, programmers may inadvertently use more of it than is safe, 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, as we have said, 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.

#### 4.4. 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 they do, and where interdependencies lie. Few problems so hinder smooth evolution of a system as hidden interactions which cause unexpected effects. Ideally, the communication between modules should be obvious from the text of both.

#### 4.5. 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. The programmer is thereby freed to think simply about what services those modules provide. While information hiding tends to decrease 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 for some reason, a module's implementation is changed, 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 (see Chapter 2 for a description) and dynamic binding. Wide varieties of related objects may be called on polymorphically, i.e., with the same interface, that captures some abstraction they share. Continuing with our example from Chapter 2, 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 software system's ability to pick the appropriate procedure for each different kind of object (bind procedure to object) as the program actually runs. The decision is made "on the fly," and it changes with different kinds of objects; in this sense it is dynamic.

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 the development 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. The same interface was used for the third-party machines as was used for the TI machines; all that was necessary was to tailor the new implementation code to the needs of the third-party machines.<sup>63</sup>

These principles of modular software construction are not easy to achieve. 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

---

<sup>63</sup> Experience report presented at OOPSLA 1992 by John McGehee of Texas Instruments.



it, you have to change the program to do it the way you should have done it."<sup>64</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 will often happen in experimenting with different solutions, it is as if one has borrowed money. Because one sloppy solution leads to problems that can be addressed with other quick fixes, the size of the debt can grow, with maintenance problems as the interest that must be paid. Eventually, and preferably sooner rather than later, the debt must be paid off, by cleaning up the sloppiness and restoring the modularity of the system, if the system is to remain evolvable. What this 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 WyCash+ 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>65</sup>

## 5. Accelerating evolution through software reuse

The implied context of discussion so far in this chapter has been the evolvability of particular software systems. We have considered what it means to be modular, and

---

<sup>64</sup> Personal interview, October 1992.

<sup>65</sup> Personal interview, October 1992.

what design characteristics tend to yield the sort of modularity that promotes evolvability of software systems, taken, implicitly, one at a time. In this section we broaden the perspective to consider an important way in which modularity promotes evolvability of the capital structure more generally: we consider not just single systems, but sets of systems that are able to share modules. Here we take up the subject of software reuse, a subject given a tremendous amount of attention in the industry today.

Software modules, when they adhere closely to the principles we have just discussed, can be reused in a variety of contexts. Increasing availability of such reusable modules, frequently referred to as software components, should substantially increase the rate of development of the software capital structure, and improve quality also. Not only does reuse reduce development costs on any particular product, but also it initiates a trend of continually increasing productivity, an upward spiral of wealth creation, as the programmers build on past accomplishments of themselves and others.

In considering reuse, we must think of software maintenance in two ways. One is what we have considered to this point: adapting and enhancing existing software systems in response to changing needs. Reusable components contribute significantly to this process, as we shall see. The other concerns maintenance of the software components themselves. Components can of course be more or less reusable as they are easier or harder to understand, or require more or less adaptation in a new setting. Maintenance of components, then, is a matter of investing in the components' reusability, by, for example, making them clearer,

simpler, better documented, more generally applicable, more modular according to the principles we discussed in the last section.

With the development of reusable components, we add another order of capital goods to the software capital structure. Software components constitute working capital for programmers, to be used in the construction of the software tools (or tool systems) they build. When they have components available, they need not build those inputs from scratch; rather they take advantage of the prior work of specialists who have built those inputs for them. Components 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.

#### 5.1. Freeing programmers to create

It has long been lamented that programmers too often build from scratch, trivially reproducing functionality that has been developed as well, or better, many times before. (Hamming 1968) Software components, particularly those based on object-oriented technologies, in providing a greater degree of modularity in programming, make reuse more feasible than in the past. With code reuse, what has been accomplished before need not be repeated, but simply incorporated, perhaps with simple modification.

Hence the most obvious benefit of software reuse: the savings that come simply from not reproducing what has been done before. This saving of programmer hours would be very significant even if the story ended here. But the programmer time and creativity that would have been spent reproducing may instead be spent creating, pushing outward the frontier of the new and challenging. This more

concentrated attention on new problems leads to an increased rate of software development overall, with the corresponding improvement in society's ability to produce new wealth.

### 5.2. Stockpiling expertise

Furthermore, the range and quality of the capital goods available to programmers steadily increase in a reuse environment. In essence, as software systems are developed, and from them reusable components are made generally available, the software capital structure grows directly. As more and more expertise is built into the environment the programmer uses, as more and more abstractions are built into reusable components ready-to-hand, the programmer may be more effective still. To the extent that these components are shared in an organization or a market, programmers stand on one another's shoulders.

A number of studies suggest the power of reusable components to augment productivity.<sup>66</sup> Sam Adams has reported on a series of products that Knowledge Systems Corp. built for Hewlett-Packard using Smalltalk, beginning with a project called Hierarchical Process Modeling System (HPMS). Adams reports that subsequent projects

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)

---

<sup>66</sup> See Tirso (1991), Ryan (1991), and Harris (1991).

Among the statistics that Knowledge Systems Corp. kept during their work for Hewlett-Packard was an estimate of reuse savings. 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 also the evolutionary cycle of ongoing development that Adams points out. Components designed in the initial project were then improved on being reused in subsequent projects. The new, improved versions were then reincorporated into the first system.

### 5.3. Generating economies of scope

Most programming today still occurs within what 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) When software development organizations move out of the project culture and begin to take advantage of software reuse, they can achieve significant economies of scope. (Teece 1980; Lavoie, Baetjer, and Tulloh 1991b) Component availability simplifies producing related functionality, typically related programs within the same problem domain. Reusable frameworks at a high level of abstraction are especially powerful, for these frameworks can form the basis of a family of related applications.

One kind of high-level reusable framework is an enterprise model. Sam Adams describes enterprise modeling as "the process of developing a software model that encompasses the nature of the business enterprise itself, its behavior, environment, and rules." With enterprise models,

a common reusable framework is designed for an entire class of applications. The functioning enterprise model becomes the reusable backbone for various applications across the enterprise, greatly reducing the complexity and redundancy that is so common in today's legacy systems. (1992c, p. 4)

High-level frameworks 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 widespread and generally used, the economies increase. At present, this kind of reuse is at best found within a few firms. But as component markets develop, we may find these kinds of economies stretching across whole industries.

#### 5.4. Reducing what programmers need to know

A consequence of widespread reuse will be programmer specialization and division of knowledge; as available components embody an increasing variety of design and domain knowledge in convenient, ready-to-hand fashion, the software industry will see ever more of the sharing of expertise across time and space that we saw in Chapter 1 to be a hallmark of economic development. Programmers will need to know relatively little about the components they use. In particular, they should need little knowledge of the implementation of established components. Their knowledge and expertise would instead concern the components' behavior – how to use them for various purposes.

For programmers, the availability of a host of excellent components embodying a great variety of functionality means not only that they do not have to rebuild the functionality themselves, but that they do not even need to be able to do so. They need not even think about how those components work, but only what they do. They are thereby freed to contribute their own special talents, insights, and capabilities to the growing body of programming knowledge. Through software capital markets – component markets, whose anticipated advent we take up in the next chapter – they are able to take advantage of, and contribute to, an extended and extending order of social cooperation (Hayek 1988) among programmers.

#### 5.5. Improving code dependability

Component use tends to decrease debugging time, as components become more dependable and error-free. As components are repeatedly put to the test in a variety of uses, their capabilities become known, and less debugging time is required. A programmer using code from a well-managed corporate library of reusable components should be able to do so with great confidence, knowing that only proven components are admitted into the library for general use.

The very techniques that make for good modularity also enhances trustworthiness. As we have seen, one of the principles of good object-oriented programming is to keep the individual elements simple, and easy to comprehend all at once. Another is to use small interfaces. The encapsulation provided by object-oriented languages also contributes to dependability. While encapsulation does not guarantee the dependability of the encapsulated component, of course, it does improve the likelihood that any problems that arise will be localized and easy to find. These

principles, while fostering reusability, contribute to code trustworthiness at the same time.

Present software reuse yields significant benefit to firms that take advantage of it. Component technology in a market setting should yield still greater benefits. While components are increasingly shared and reused within firms, there is still little reuse across firm boundaries. To economists sensitive to the powers of markets to discover and communicate knowledge, the prospect of component markets is exciting. Reuse within a market setting will yield enormous productivity gains, by disseminating widely the most effective technology. This prospect is the subject of the next chapter.



## Chapter 5

### Evolving the Capital Structure: Markets for Software Components

*Saw the heavens fill with commerce, argosies of magic sails,  
Pilots of the purple twilight, dropping down with costly bales;  
- Tennyson, "Locksley Hall"*

*Economics has from its origins been concerned with how an extended order of human interaction comes into existence through a process of variation, winnowing and sifting far surpassing our vision or our capacity to design. ... Modern economics explains how such an extended order can come into being, and how it itself constitutes an information-gathering process, able to call up, and to put to use, widely dispersed information that no central planning agency, let alone any individual, could know as a whole, possess, or control.*

*- F. A. Hayek<sup>67</sup>*

#### 1. Introduction

In this chapter we take a different perspective on the nature of capital structure development. We move beyond what we may call social learning in the small – the learning necessary to develop and evolve particular capital goods – to consider what we may call social learning in the large – the development of new institutions, understandings, and practices that support the capital structure and allow it to grow

---

<sup>67</sup> (1988, p. 14).

more rapidly. In particular, we look at the prospect of markets for software components. (Cox 1990 and 1992; Lavoie, Baetjer and Tulloh, 1991b and 1992)

Many believe that increasing software reuse within particular firms will boost significantly the productivity of those firms, in time to market, quality, maintainability, and range of products offered. Even if these hopes come to be fully realized, those productivity gains may be only the embryo of the benefits possible for software reuse. Far greater gains will result when and if reuse is extended across firms through component markets. Improvements in modularity, and especially object-oriented technologies – go a long way to make possible vigorous markets for software components. Interestingly, while object technologies clearly make component markets possible, this seems to be an unintended consequence; most of the designers and developers of object technologies have had other things in mind.

Extensive software component markets, on the verge of which we seem to stand today, should enhance social learning and therefore wealth creation in a number of ways: Markets will make possible a dissemination of the knowledge embodied in software components far beyond what is possible in the absence of markets. Also, markets should support a more rapid development of new knowledge of this kind, through extending the number and diversity of people involved in the learning dialogue, and through extending the dialogue deeper into the structure of production.

In order for software component markets to flourish, however, more social learning, in the form of development of supporting institutions and attitudes, is necessary. The most significant needed change, which we take up in detail in section 4 below, appears to be improved means of pricing components, based on new property rights

institutions. The very characteristic of software which makes it so suitable for this study – its being primarily knowledge, largely independent of physical embodiment – has made software problematical for those who produce it. Software can be copied at almost no cost, into other machines, onto diskettes, over networks. For smaller units of software such as small-scale software components, this presents a real problem: it is very difficult for the producers of such components to be paid adequately for the benefits they offer; it is too easy to get a copy of a component without paying for it. Accordingly new means of establishing and securing property rights in software need to be developed.

Also standards must be evolved to improve complementarity of different components. New means of distributing software will have to be developed and accepted. All these changes will require cultural shifts. And of course, all these changes will impact one another: components, component markets, and the institutions that support them will co-evolve.

The development of component markets, then, will be a matter of social learning in the large, not constrained to the limited settings we have considered thus far, of certain clients, designers and single software applications into which their knowledge is built. Our context in this chapter is the software industry as a whole, (overlapped as it is with many other industries, of course). As software systems embody and re-present in useful form a large amount of knowledge from many people, far exceeding what one person could know, so likewise the systems of interaction we know as markets embody and present to us in useful form a great and various (and evolving) body of knowledge. As the development of software is a social learning process, so also is the development of software component markets.

## 2. Component markets as an unintended consequence of improved modularity

"For want of a nail, the shoe is lost; for want of a shoe, the horse is lost; for want of a horse, the rider is lost," writes George Herbert.<sup>68</sup> Small developments can have great consequences. This seems to be the case, though in a happily positive direction, with object technology. Object-oriented programming languages were developed initially to facilitate computer simulation and to empower computer users to accomplish their various purposes better. (Goldberg 1981) As we have seen, over time object-oriented technologies have been recognized as useful in enabling rapid product development, reducing maintenance problems, and facilitating code reuse. Aside from a few far-sighted individuals however, few have seen clearly that improved modularity through objects has still another benefit to offer: the potential to revolutionize software development, by enabling thoroughgoing specialization and division of knowledge, mediated by markets. (Lavoie, Baetjer, and Tulloh 1992) The attention now being paid to reuse, and to constructing well-defined software assets suitable for reuse within a firm (Adams 1992a) inevitably leads in the direction of component markets, because the more understandable and complete is some software component, the more likely it is to be desirable to users outside the firm in which it is developed.

Component markets were possible before the development of object-oriented technologies only to a very limited extent, because previous technologies involve

---

<sup>68</sup> Jacula Prudentum, 499.

too many interdependencies: previous "components" have rarely been truly separable and independent. Previous technologies do not facilitate what Meyer calls modular composability and modular understandability. With the encapsulation of data and functions that objects provide, however, it is possible to build "computers within computers," units of functionality that make sense on their own, and can be incorporated into a variety of different systems. With this capability, it is now possible to produce meaningful units of functionality that can be combined in a variety of ways. These meaningful units of functionality are sellable units, providing, of course that the problem of property rights and pricing is solved. Hence distinct software modules make possible markets through which they may be widely disseminated, and thereby free programmers from the need to redevelop such functionality on their own.

When and if software component markets develop, they will constitute a significant further enrichment of the complex pattern of complementary relationships that is the capital structure. Complementarity continues to be of the essence: the new, finer-grained elements of the software capital structure must work with one another to be valuable (and hence to be capital). In the market context, however, complementarity will be mediated more by market forces than by direct planning, as it is within a firm or project. Whereas within a given project, one might ask, "what does the interface of this object need to be, so as to fit with the objects my colleagues are building?", in the market context one needs to ask, "what does the interface to this component need to be, so as to fit with the conventions and standards that are evolving in the marketplace?"

In the present context, then, we are looking beyond the learning necessary to build a software system successfully, at the learning necessary to develop the social system – the market, what Hayek called the extended order of human cooperation – so as to achieve a general increase in wealth.

### **3. Learning through markets**

Extensive component markets will yield important benefits that will transform programming practice for the better. The transformation will be profound. Consider the current state of division of knowledge in 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 building contractor's being asked to build a new house, and beginning by going out to the forest with a chain saw to cut lumber for two-by-fours 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. This picture seems to us absurd and wasteful. But there was a time not long ago when homesteaders did exactly this. Only the development of widespread markets for housing components has made possible our present division of knowledge and labor with their multiple stages of production, and the efficiencies and higher quality that result.

Of course, with intra-firm reuse, the picture improves. It roughly parallels a situation in which the building contractor has certain grades of lumber and wrought

iron on hand from previous jobs, which can be incorporated into new buildings with little or no adaptation.<sup>69</sup> This is a great advantage. Nevertheless, it falls far short of what that can be achieved through extensive specialization and division of knowledge made possible by market relationships, in which two-by-fours, roofing shingles, bathroom fixtures and the rest of the materials are built by specialists, with the house builder specializing in assembling the parts to specification.

Software component markets offer this kind of extensive specialization. They promise a number of benefits in generating and making good use of the knowledge that exists in the software development community, but which is in large part trapped within particular firms. More important, they promise to elicit a vast amount of additional, latent knowledge that will be forthcoming when there are market structures to support its discovery and exploitation. Almost undoubtedly, again providing that the pricing and property rights issues can be resolved, there will come a time when the structure of production of software is just as specialized as that of house-building, and we will look back on present practice as just as primitive as the house-building practices of the old frontier.

### 3.1. Knowledge dissemination

One of the most obvious benefits of component markets is that they will allow a far greater number of software builders to take advantage of any particular body of

---

<sup>69</sup> Of course the analogy is not perfect. Software products are not perishable, so once you have built a software two-by-four, you always have that item available. The challenges to software reuse have to do with such matters as locating, understanding, adapting, and testing the component, all with enough ease that it is simpler to reuse than to rebuild.

embodied knowledge that may be offered for sale. Market incentives will encourage component vendors to find those development organizations that need the components they can provide. Instead of being stuck within the confines of a single firm, reuse can spread across firms. What has once been accomplished well need not be replicated, not within the firm that accomplished it, nor any other firm.

Of course there will be trade secrets, and often firms will choose not to release the components they have developed for sale to the general public. But as long as any given kind of functionality is generally needed, there will be an incentive for some independent component supplier to try to produce and market it.

### 3.2. Specialization

In considering the benefits of internal reuse in the last chapter, we mentioned that reuse should reduce what programmers need to know, and, in freeing them from reproducing functionality, allow them to devote their attention to developing new functionality. In short, component markets will allow programmers to specialize more. Some may specialize on building components, some on assembling applications with those component.

This division of knowledge and labor itself improves learning, because specialists are able to develop a more thorough understanding of and expertise in their chosen problem areas. One of the great challenges of writing good object-oriented software is drawing the best possible abstraction boundaries between the different elements of the system being modeled. It can take a long time to develop enough familiarity with a particular problem area to discover how these abstraction boundaries had best be drawn. Specialization will allow this kind of learning and



discovery. Under present conditions in industry, with software being applied to ever more particular and specialized functions, this kind of specialization would seem to be a great benefit, because much of what the programmer needs to concentrate on is not programming skills as such, but the detailed and changing needs of the field for which he is writing software.

One kind of knowledge we would expect component specialists to build into the components they market is the knowledge of what kinds of customization will be needed for their components, and how to make that customizing easy for the downstream programmers who will incorporate the components in specialized applications.

### 3.3. Information hiding through separating the stages of production

In discussing modularity above, we spoke of one of the benefits provided by information hiding: it prevents programmers from concerning themselves with how an object is implemented, and thereby from introducing any problematic dependencies based on that implementation. They know only what the interface is, and what services that object provides. With the evolution of increasingly distinct components built to be used in different stages of production, we have something similar to information hiding, and potentially more powerful in improving the quality of software development.

George Bosworth, the chief technical officer of Digitalk Corp., points out that at present, virtually all programming is done with the same set of tools.<sup>70</sup>

Programmers do the same kinds of things with the same kinds of tools whether they are writing a small algorithm or a large application: they read and write code. In most cases, almost all the code is directly available to them. This, Bosworth suggests, is a problem. Reuse will happen when the techniques used to reuse the components differ from those used to build them. He holds that for programmers to be able to see the code of the components they are using is problematic. It draws attention to how they were built, and away from how may they be used.

(Additionally, it makes possible the perilous business of revising and "improving" those components, whether consciously or inadvertently, with all its problems of introducing inconsistencies and bugs, and violating the expectations of other members of a programming team.)

With this kind of idea in mind, Digitalk has built a product called PARTS, the Parts Assembly and Reuse Tool Set. PARTS offers a platform for truly distinct stages of software production. With PARTS, the user of a particular component does not and cannot see how it is built.<sup>71</sup> Thus the user's focus is necessarily on what he or she will use it for. Irrelevant detail is suppressed. The user assembles applications using the PARTS Workbench, by linking various components together visually on

---

<sup>70</sup> Personal interview, November 1991. I am indebted to Mr. Bosworth for my appreciation of this point, and my understanding of its importance.

<sup>71</sup> This is known as a black box component. While there have been black box components available in other settings before, including reuse programs in large firms (Tirso 1991, Harris 1991, Prieto-Diaz 1991), Digitalk's PARTS is the most important commercial platform for black-box components to date.

the screen, occasionally dealing with a limited amount of code. Digitalk is actively encouraging third-party developers to build a variety of components for the system, offering all sorts of special functionality, which will then be available for sale to other users of the PARTS Workbench for assembly into applications.

The development of PARTS and other systems like it may stimulate an important step in the evolution of the software industry: the development of a new stage of production. This stage is suggested by a minor confusion in the terminology being used to discuss PARTS. Up until now, one has been able to communicate fairly clearly speaking only of programmers and of users. Programmers were those who build software. Users were "end users" – the people who make use of the software applications. The limited terminology suggests what is largely true – software applications are built at a single, very complicated stage of production. Or rather, given that the work of translating the finished code into machine language is done by compilers at a separate stage in the process, we may say that applications are built in two stages.<sup>72</sup>

But in a recent article discussing PARTS (Bosworth 1992), when George Bosworth speaks of users he does not mean end users. He means those who will (re)use PARTS components in assembling applications. This is programming, although of a different sort than what we are accustomed to, since programming with PARTS mostly involves visual tools rather than coding, and since one who programs with PARTS is equipped with a new and rich sort of working capital, ready at hand.

---

<sup>72</sup> Of course the compilers in many languages operate automatically, without human intervention; that is, at the compiling stage human programmers are present only in the form of their knowledge, embodied in the compilers.

Hence in this context we have three kinds of programmers: those who build compilers and programming languages, those who build components, and those who use components to build applications. We might expect that as the trend to component markets and component assembly systems such as PARTS continues, new terms will evolve to capture the distinction.

Of course this division of software production into stages of production can potentially go on a long way, with small components being built into larger components, and these into still larger components, in an indefinitely long progression. Indeed, the PARTS Workbench provides the capacity to build new, larger units out of a combination of smaller PARTS (Digitalk calls them nestparts or subassemblies), and treat these new entities as separate, independent PARTS. Presumably, we can expect this division of labor and knowledge to be limited, in the end, only by the extent of the market, and the market will be very large indeed.

It may be that we are seeing the beginning of another transformation of what it means to program, similar to the shift that occurred when higher-level languages were built which could automatically do the low-level "programming" into machine language. Perhaps, with the embodiment of a wide variety of programming knowledge in a broad selection of readily available and easily combinable software components, virtually everyone may become a programmer. There was a time when telecommunication was demanding – only trained telegraph operators could effectively communicate with one another across long distances, because the available tools required special knowledge to operate, and communication had to be in Morse code. But in time better, handier capital goods for telecommunications were devised. In particular, the telephone was invented. Now everyone may be a

telecommunicator, simply by speaking into a telephone, whose use is natural, easy, and almost self-evident. There seems to be no reason why "programming" should not become as easy, as we learn how to build the requisite knowledge into better, handier programming tools. As telephones let us move from telecommunicating in Morse code to telecommunicating in natural language, new programming tools may let us move from programming in code to "programming" in natural language.

### 3.4. Market learning

A crucial benefit that component markets will give software is more extensive and detailed market learning. As Hayek has pointed out, the market process is a discovery procedure (1978) through which market participants may learn what is needed and wanted, what is available, and where opportunities lie. Of course all we have discussed to this point assumes a market context. The point is not that component markets will add something different, but that they will extend market processes more deeply into the software development process, and thereby deepen and enrich the learning that can occur.

Obviously market feedback drives software development. The public's desire for certain features in word processing or spreadsheet packages – whether registered through direct praise, complaints, published reviews, or simply changes in market share – directs the subsequent development of the applications. The same kind of thing is true even with large applications built in-house for large firms. The users of the application are in effect the customers of that firm's programming team that builds and maintains the application, and the users' satisfaction or dissatisfaction with performance and features will shape what the programmers do next. This iterative, back and forth sequence of the software maker's offering a new release of

the product, and the public's responding to it with market feedback, is another instance of the dialogue-like process we saw occurring with prototyping. It is an important source of knowledge about what is needed and wanted: the knowledge gained can then be embodied into the next release of capital goods in question.

An especially salutary aspect of this process is the knowledge generated by the multiple experimentation that occurs with competition. In a competitive environment, different providers try different solutions to a problem, essentially offering them for approval to their customers. The better solutions tend to become known and widespread. Furthermore, the very variety of attempts is suggestive of what else may be done: sometimes failed attempts give the observers ideas as to how some aspect of that attempt might be successfully used. With competition also comes an added dimension to the dialogue between providers and customers; that is what A.O. Hirschman calls exit, the option that customers have of simply leaving the dialogue – taking their business elsewhere. Of course taking this option sends a strong signal to the providers that they are somehow falling short.

Component markets will make all these kinds of feedback finer-grained and more extensive for the software industry, thereby generating more knowledge in the system. Not only whole applications will be judged and commented on, whether directly or through exit; but now the component building blocks will be subject to the same kind of dialogue and discovery. The effect should be to improve the rate of improvement in the software capital structure at all levels.

## **4. Aspects of component market evolution**

While the promise of component markets is great, and while object technologies make possible the building of reusable software components that may be bought and sold, some substantial changes must occur in the software industry and the software development culture before component markets can flourish. (Lavoie, Baetjer, and Tulloh 1992) These developments constitute social learning in the large: the evolution of a body of shared assumptions and practices.

### 4.1. Development of standards

One of the main obstacles to software component markets is the lack of standards. Even though the different object-oriented programming systems all allow the construction of reusable and potentially sellable components, in most cases these components cannot be integrated without an effort far exceeding what it would take simply to replicate their functionality. A main problem is the incompatibility of objects built in different languages. Objects built in Smalltalk cannot be incorporated into a C++ program. Worse yet, there are incompatibilities among the different class libraries developed for the same language (class libraries are sets of objects, usually sold as a package, offering a variety of functionality). While there is a variety of class libraries to choose from if one uses, say, C++, one pretty much has to choose, because the libraries will not work together. For example, the same class name might be used in two different libraries for two entirely different classes.

Lack of standardization fragments the potential market excessively, thereby reducing the incentive to develop components for sale.

There are, however, promising developments on this front. IBM is developing what it calls a System Object Model (SOM), which is intended to allow objects written in different languages to work together. Not only that, it allows classes of objects from different languages to be adapted (the technical term is "subclassed") by the users as needed, without any knowledge of the original language required. Hence the System Object Model provides a bridge between languages.

Another development comes from the Object Management Group, a consortium set up to establish standards for sharing components across networks. The Object Management Group has already established the Common Object Request Broker (CORBA), a standard for object interaction that is gaining widespread acceptance among some of the largest software vendors.

Digitalk intends for PARTS, which is built in Smalltalk, to provide the capacity to "wrap" objects built in other languages. This will allow component developers using other languages to build PARTS components. Also it will allow companies with a large investment already sunk into components built in other languages to transform them into PARTS components, which can then interact freely with other PARTS components.

For any of these different systems to become established as a standard around which component markets grow, an adequate number of industry participants must embrace it, learning its virtues and defects, and how it can accommodate their needs. Importantly, this is a coevolutionary matter: among the most important



things any market participant must learn about an emerging standard is that it seems to be accepted by others. The network externality here is substantial.

One would hope that a number of different standards emerge. Different kinds of standards will be appropriate for different purposes. Further, the competition among standards is itself a valuable social learning process. Much can be learned through comparing the advantages and disadvantages of competing standards for different purposes.

#### 4.2. Pricing

As we have suggested, even if there presently existed a number of generally accepted standards for component interaction, markets for components might not flourish, because current methods of pricing software are probably inadequate to the special demands of component markets. With today's pricing institutions, it might be very difficult for component producers to be paid adequately for what they produce. Software is easy to copy; indeed, the cost of copying a program approaches zero. Under today's pricing institutions, this fact is a problem obstructing component markets: a component producer might conceivably build a valuable component, sell a few copies, and then receive no more revenue, even though her component is widely copied and widely used. New methods of pricing, based on new conceptions of property rights to software, will need to evolve. Under appropriate pricing institutions, the ease of copying might be turned to a benefit for component producers, and accelerate the development of component markets. Property rights structures continually evolve (Mackaay 1990), and they undergo significant transformation in response to changes in technology (Palmer

1989). As technologies develop, the societies using them must learn what kinds of property rights structures work well.

Vendors of software need assurance that they will be rewarded for the value they provide others; they need protection of their property rights. At present, this protection is afforded, imperfectly, by licensing and copyright. Nearly all software is licensed, not sold. For mass market software, the courts have evolved a system of shrink-wrap licensing through which, when a buyer breaks the shrink-wrapped seal of a software package, he is thereby agreeing to the terms of the license. A variety of licensing arrangements is being developed for use on networks: In some cases different prices are charged for different numbers of users; in others the network is equipped with a metering system that allows only a limited number of uses of an application at a time, in the manner of a lending library with only a limited number of copies of a book for check-out. Furthermore, software may be copyrighted: one may not legally make more than a very few copies (e.g. for purposes of backup) under the fair use doctrine.

Of course, these legal restraints do not work perfectly. The almost effortless ease with which software may be copied – a marvelous characteristic from the standpoint of what it means for the spread of knowledge capital – is, in the context of present legal and market institutions, a severe liability. Because software is presently sold by the copy, ease of copying is a problem. "Piracy" – illegal copying of software – means lost revenues to software producers, and hence a reduced incentive to produce it. The problem is worse with software components: because they are smaller and less expensive, it is more difficult to detect the copying of them, and it is uneconomical or impractical to secure revenues by such strategies as

bundling them with documentation or the promise of upgrades. There is also the problem that incorporation of too many third-party components into an application can push its price too high: Each component vendor, fearing copying, may be induced to charge a fairly high per-copy royalty on his component. The application vendor must of course cover these costs in the price of his application. It is easy to see that incorporation of too many high-priced components can price the application out of the market altogether. Hence application vendors have the incentive to rebuild functionality rather than buy it.

Some have suggested, persuasively, that adding a new, substantially different kind of pricing option can greatly facilitate the emergence of component markets. (Miller and Drexler 1988b, Cox 1992) This approach is to allow users to pay for software by the use rather than by the copy. It is known as charge-per-use, pay-per-use, or superdistribution. (Mori and Kawahara 1990) Under a charge-per-use system, a meter of some kind in the underlying operating system would keep track of how much certain software and software components are used,<sup>73</sup> and the user would be charged accordingly. Some means (there are a number of alternatives) would have to be settled on for ensuring payment, which would probably be handled on a monthly or quarterly basis through a clearinghouse, which would distribute payments to the different vendors. (The statement could detail usage for the customer in the manner of a telephone bill.)

---

<sup>73</sup> How usage might be defined is an interesting question. Some vendors might charge by time of use, some by number of uses, etc. The different methods can coexist. Presumably market experimentation will reveal which techniques are best attuned to which circumstances.

Charging by use rather than by the copy allows vendors of software and software components to segment their markets on the basis of their customers' intensity of use. Occasional use of a very expensive program would become feasible under charge-per-use, and vendors would have a better chance to be paid by intense users in accordance with the value they actually receive from certain applications.

By providing a technological means of ensuring payment, charge-per-use eliminates many of the current problems of enforcing contracts by monitoring and by legal procedures. Indeed, in a charge-per-use system users would be encouraged to copy their software freely and distribute it widely, to friends, co-workers, and others.

From the standpoint of component markets, charge-per-use has the great advantage of giving component producers reasonable assurance of payment. Freed from the worry that components they build will be illegally copied and widely used, with little reward to themselves, potential producers of components are likely to become actual producers of components. For the same reason, component producers would feel free to charge, for use of their components, only a small fraction of what it cost them to build those components; they would reasonably expect to be paid for their effort incrementally over many uses. This low per-use price of the components would in turn mean that prospective users of those components would be willing to incorporate a number of third-party components into their own applications, because doing so would not drive up the per-use cost of their applications too much.

There is an important social learning advantage to charge-per-use. That is, the underlying system would be able to collect extensive valuable information about the nature of usage. Figuratively speaking, it would allow the market dialogue

among vendors and users to be more rich and detailed, so that they may come to understand one another better. This finer-grained market feedback would inform subsequent software development, resulting in lower costs to producers, and better quality to users. At present, application providers do not have much information from their users as to what aspects of those applications are used the most, or most valued. But with a charge-per-use system, application and component developers could gather detailed information of this kind. They would then have a better idea which modules to enhance first, which modules to deemphasize, which to improve in performance, etc. Significantly, because different users use applications in different ways, detailed usage information would make it possible for vendors to customize particular versions to the needs of different customers (somewhat in the manner that telephone companies today offer different packages to users with different intensities of use). It would not even be necessary for particular users to know which version of a software application they are using; the vendor could simply monitor their use, and customize their packages accordingly.

Privacy issues arise with this technology. Some may not want anyone to know how much they use different (parts of) software applications. Encryption technology exists, however, for allowing precise data to be collected, charges made and royalty payments paid, without anyone being able to tell who used what. Accordingly, those who wanted privacy could have it. On the other hand, many will probably want the advantages that come from their software suppliers having good information about their usage.

### 4.3. Distribution channels

Software component markets will require new distribution channels. Current software distribution channels, generally expensive and aimed at the mass, end-user market, are ill-suited to components, which require inexpensive channels aimed at developers and sophisticated end-users. A small-scale software component, for example, that may sell for, say, \$80 to each of a thousand potential users nationwide, cannot afford \$100,000 worth of packaging, marketing, and distribution costs. The industry needs to develop affordable means by which producers can easily distribute their components, and users can easily access them.

Fortunately, complementary technologies are being developed. In particular, electronic distribution seems very promising, especially if charge-per-use is enabled. Components may be easily loaded onto telecommunications networks, and downloaded by potential users at very low cost. On-line cataloguing of components can lower the costs of communicating what components are available and what they do. Additionally, electronic marketplaces can reduce the transaction costs of buying and selling components through electronic payment and maintenance of accounts.

In January of 1992, an electronic marketplace providing the above services came into being. The American Information Exchange (AMiX) opened an electronic market for software components and consulting.

Components can be inexpensively stored on the system and downloaded by buyers for immediate use. To facilitate custom development the system supports small-scale consulting with negotiation, contracting, and delivery on-line. AMiX handles all billing and accounting centrally, freeing market participants from accounting overhead. On-line charges

are at cost, and in any case the system allows users to do most of what they need to do from their local image of the system, connecting only for short periods. (Baetjer and Tulloh 1992).

An alternative means of low-cost component distribution is CD-ROM (compact discs containing read-only memory). This technology allows very inexpensive distribution of vast quantities of information.

#### 4.4. Cultural shifts

All of the technology necessary to support charge-per-use software markets exists. The advantages of such markets are arguably great. Why, then, do we not have charge-per-use markets. One important reason is that while such markets are familiar to our culture – we buy telephone service, electricity, some television, water, etc., by the use – there is resistance to charge-per-use in some parts of the programmers' subculture. Cultural shifts in general are an important aspect of market evolution. A shift in culture constitutes a significant amount of learning about shared expectations.

Regarding charging per the use of software, rather than per the copy, there is the particular difficulty that it reminds some programmers of the "bad old days" of time sharing, when they were charged for scarce, precious computer time. Those who believe charge-per-use software to be a return to time-sharing need to be reassured that this is not the case. Charging per use need only be an additional pricing option, fully compatible with pricing per copy, which will not disappear. Those who advocate charge-per-use must propagate their ideas widely, explaining how it can work and what its advantages are.

An important cultural shift that would seem to contribute well to the prospects for software component markets is a change in the style of teaching in software engineering schools. Commonly, students are taught to approach problems from scratch, devising their own solutions to problems that have been solved by hosts of other students and practitioners before. While this sort of practice has its place, students of software engineering also need to be taught the benefits of software reuse, encouraged to make use of industry-standard components, and trained to make use of the prior work of others. In this respect, the software engineering schools would do well to start their students with object-oriented programming as the current best style of programming, instead of teaching introductory courses with traditional languages and then introducing object-oriented programming as something new and out of the mainstream. Once one has learned traditional approaches to programming, it is more difficult to change one's mindset to the object-oriented way of thinking. There is no point to teaching students bad habits, and then asking them to unlearn them. As industry moves more and more to object technologies, we can expect this shift to occur.

Whether in the schools of software engineering or in practice in industry, 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 reuse by others, making it clear, understandable, modular, and well-documented.

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 developers 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. This will necessitate a change in accounting: the costs of developing reusable assets must be spread over many projects; 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 reuse. Managers today, with such specific milestones to meet, are understandably unwilling to permit the development of reusable objects, if doing so puts them over budget on the project for which they are responsible. High-level management must recognize that developing reusable software assets is an investment in future productivity that deserves their support.

## **5. Summary**

The possibility of widespread markets for software components is a consequence, mostly unintended, of the improved modularity of software made possible by object technologies. Component markets would foster a substantial enriching of the capital structure, with greater specialization, division of knowledge, and resultant embodiment of useful knowledge in working capital for programmers – software components. Truly separating different stages of production of software would also foster specialization, and allow those who build applications by composing various components to focus on the problem at hand, unconcerned with how the components they are using were built. Component markets would extend the benefits of market feedback and market learning beyond whole applications to the components of which software is built.

For component markets to emerge, however, a significant amount of social learning is necessary. Standards must be evolved to enable disparate objects to work together. Better distribution channels, such as electronic marketplaces, need to be developed and used. New property rights structures and pricing methods must be developed to take into account components' small size, ease of copying, and potential composition in large numbers into final applications. Charging by the use rather than by the copy is a promising possibility. To support the emergence of component markets, a variety of cultural shifts is necessary also, on the part of programmers and managers. They must come to accept widespread reuse, with its implications for sharing one another's work and developing with a series of projects in mind.

Once component markets have evolved, what might the next major development be? (We would not expect the evolution to stop, of course.) Miller and Drexler (1988b) discuss a fascinating possibility. They suggest that through encapsulation, objects give programming the same kinds of benefits that property rights give economies. Why not, then, seek to incorporate more aspects of markets into programming systems? They suggest the further market-oriented development of constructing objects which bid for one another's services in, thereby giving programming the benefits of price information about relative scarcity. Such systems would probably be self-contained at first, with the different objects in a program negotiating with one another in terms of an internal, virtual currency. But with charge-per-use implemented across a network, there would appear to be no reason why an object on my machine should not be able, eventually, to bid for the services of objects (and other computational resources such as CPU time) on other machines. Such distributed, market-based systems of computation are what Miller

and Drexler call "open agoric systems." Their implications, not least for very rapid market-based discovery, are profound indeed. (Consider: properly programmed objects could carry out a large number of lengthy, multi-party price negotiations in microseconds.)

## Chapter 6

### Conclusions: Implications for Economic Development and for Growth Theory

*Not in vain the distance beacons. Forward, forward let us range,  
Let the great worlds spin forever down the ringing grooves of  
change.*

*Through the shadow of the globe we sweep into the younger day;  
Better fifty years of Europe than a cycle of Cathay.  
- Tennyson, "Locksley Hall"*

#### 1. Introduction

In this brief concluding chapter we broaden the perspective greatly, and consider the implications of our findings to the economy as a whole. What are the implications of what we have discovered about software development for the development of the economy as a whole? And what can that tell us about economic theory? Before turning to these questions, we need to establish the applicability of the concepts we have been discussing to tools in general – hard tools as well as software tools.

#### 2. Applicability to hard tools

We chose, in this inquiry, to focus on software development, because with software the knowledge aspects of capital goods are so immediately apparent, and the physical aspects are so much in the background. This has allowed us to focus on

capital goods as embodied knowledge without being distracted by steel and glass and silicon and ceramics, and the important challenges of embodying design knowledge in those physical substances. In this section we verify that the issues of social learning and system evolvability, which we found to be crucial in software, are also fundamentally important in hard tools. The same issues apply whether we are talking about designing and producing a new word processor or a new hammer.

### 2.1. Prototyping and social learning

The key concept we explored in Chapter 3 is that the development of new capital goods is a social learning process. It is a learning process because it is a matter of embodying knowledge, and it is a social process because it calls on the knowledge of a variety of people, which is embodied in a form that is available for shared use. The knowledge is dispersed, incomplete, and often tacit. The proof of the point we found in the nature of the processes and tools used in initial software development. Chief among these are rapid prototyping and a variety of tools and methodologies for managing the complexity of the design process. Do we see the same kinds of processes and tools used in the development of hard tools?

We do. Prototyping is particularly important in manufacturing. Steven Wheelwright and Kim Clark address the development of physical goods in their recent book Revolutionizing Product Development. (1992) They argue that

...prototyping and its role in design-build-test cycles is a core element of development and a major area of opportunity for managements seeking to improve the effectiveness and efficiency of their development process. (p. 260)

They focus in particular on "[i]ncreasing the rate and amount of learning that occurs in each cycle." (p. 260, emphasis added)

New, computer-driven devices for the rapid prototyping of physical tools and parts<sup>74</sup> are being employed to great advantage by auto makers, aerospace companies, and tooling companies. (Chaudry 1992) These new prototyping tools are much faster and less expensive than conventional techniques, providing more rapid and frequent feedback to designers and prospective users.

The purpose of prototyping hard tools is the same as for software: to elicit information from the different people whose (often tacit) knowledge can contribute to the design process.

Because even simple prototypes can convey substantial amounts of information, they serve as a bridge between individuals and groups with very different backgrounds, experiences, and interests. Thus management can use prototypes to gauge, share, and extend organizational knowledge. (Wheelwright and Clark 1992, p. 274)

As with software prototypes, physical prototypes serve as the vehicle for dialogue through which new knowledge is elicited and understood by the various participants:

The physical object represented by the prototype becomes the vehicle by which different contributors can focus and articulate their concerns

---

<sup>74</sup> These devices use such techniques as hardening liquid polymer with an ultraviolet laser. The laser is guided by computer-automated design (CAD) drawings of a series of cross-sections of the tool to be modeled. Layer after layer is deposited as a computer controlled lift lowers the emerging model into the liquid. Models can be used as the prototypes themselves, or as molds from which the actual prototypes are cast.

and issues, and reach agreement on the best ways to resolve conflicts and solve problems. (Wheelwright and Clark 1992, p. 273)

The physical nature of the prototype makes it more understandable to those whose own knowledge of it is more tacit than articulate. Communication through a prototype often succeeds better than communication through symbolic representation: Through rapid prototyping, Alcoa has not only shortened its manufacturing review process substantially, but also has "minimized mistakes caused by misinterpretation of manual drawings and prints and miscommunication of design details." (Chaudry 1992, p. 78)

## 2.2. Modularity and evolvability

In Chapter 4 we explored design evolvability through modularity. Not surprisingly, modularity is very important in the design of hard tools also. A concept currently important in the engineering literature is "design for manufacturability"<sup>75</sup> (DFM), in which modularity and component assembly are important. The design for manufacturing literature discusses specific modularity issues closely related to those we saw raised by Bertrand Meyer. Design for manufacturability addresses understandability (regarding, e.g., whether a part is symmetrical or not) and the nature of interfaces (ideally they should be simple enough so that parts fit or snap together and assembly tools are not required). Another important issue is standardization, for precisely the same reasons it is important in software: standard

---

<sup>75</sup> For representative work, see Shina (1991) and Suh (1990). "Knowledge-based, object-oriented" computer-automated design systems and their use in design for manufacturability are discussed in Belzer and Rosenfeld (1987), and Cinquegrana (1990).

parts are easier to reuse in different, but similar designs; they are more reliable because tested in a variety of uses; they are less expensive to use because they do not need to be tested; and they are more likely to be reused rather than replicated because they become generally known. (Kamm 1990)

While the design for manufacturability approach generally stresses the importance of modularity to the manufacturability of particular, single products, Wheelwright and Clark take pains to establish its importance to what they call producibility as well. What they mean by producibility is what we have been discussing as evolvability. They urge manufacturers to think beyond designing single products, and think instead of "an approach to design that comprehends the product family as a whole." (1992, p. 237)

Given increasingly fragmented markets and the need to offer specialized products that meet the requirements and demands of increasingly diversified customers, [manufacturers] need the capability to produce a high variety of products at low cost. Moreover, [they] need to be able to respond effectively to shifts in the product mix that occur from time to time in unexpected ways. (1992, p. 237)

In the terms we have been using, any design or family of designs will have to evolve as conditions change, and what changes will occur is uncertain. Therefore it is well for the designs to be evolvable. And evolvability, in hardware as well as software, depends on modularity of design. Wheelwright and Clark speak in familiar-sounding terms:

In the case of our gear design problem, a firm using modular design would not design a new automatic rewind system every time it brought out a new version of a particular camera. Instead, the project to develop the platform product would include an effort to develop a new rewinder and a new gear system that designers would use in several future versions of the product. Engineers working on the platform would design the rewinder to fit a given space constraint and would establish



interfaces (how the parts fit together physically, how control is achieved, how the users interact with the rewinder) to guide future development efforts. (1992, p. 239, emphasis added)

The principles we have uncovered with respect to software design, then, seem to be applicable to hardware design. Again, we have not the opportunity to explore them in any depth here, but such exploration seems fruitful.

### 2.3. From modularity to component markets

In Chapter 5 we suggested that the improvement in software development technique represented by object-oriented languages enables the construction of reusable software components, and hence lays the groundwork for markets for components and thence to a new software component industry. Component markets will emerge, we argued, through a substantial amount of social learning, in evolution of standards, in development of new means of distribution, in changes in cultural attitudes and practices, and, probably most important, in the development of new approaches to pricing and property rights in software. We argued, further, that this new industrial structure, with an increasing number of distinct stages of production, would make possible significant social learning about what sort of software is needed, and by whom.

Clearly the development of modular systems has followed a similar course in a number of industries that produce hard tools. Richard Langlois and Paul Robertson have documented the evolution of modular systems in the stereo component and microcomputer industries. (Langlois 1990, Langlois and Robertson 1991, Robertson and Langlois 1992). Development of distinct modules did lead to markets for components; these markets depended for their vigor on establishment of standards;

new channels of distribution evolved. And Langlois and Robertson point to clear benefits of this evolution:

We argue that [modular] systems offer benefits on both the demand side and the supply side. Supply-side benefits include the potential for autonomous innovation, which is driven by the division of labor and provides the opportunity for rapid trial-and-error learning. Demand-side benefits include the ability to find-tune the product to consumer needs and therefore blanket the product space more completely. (Langlois and Robertson 1991, p. 2)

The pricing and property rights issue appears to be fundamental in software, and an area of real difference from physical capital goods. Markets for capital goods, and the benefits they confer, depend on entrepreneurs' ability to buy and sell the capital goods successfully, with producers capturing in profits some of the benefits they provide. Software is very different from physical capital goods in being mostly knowledge, easily distinguishable and separable from the physical media and computers in which it may be loaded. The peculiar nature of software as a hybrid – neither pure knowledge nor hard physical good – means that it cannot be bought and sold in the manner of nuts and bolts, computer chips, or automobile fenders. Some new pricing procedure, based on a new property rights arrangement, appears to be necessary for vigorous software component (capital) markets to emerge. One possibility is distributing software widely at no charge ('superdistribution'), and charging by the use rather than by the copy.

### **3. Implications for economic development: exponential growth?**

What conclusions can we draw from this inquiry about the rate, and more importantly, the potential rates, of economic development, given appropriate conditions? Because economic development is in large part a matter of the "complexifying" of the capital structure – the on-going enrichment of the capital structure as new, ever more specialized knowledge is developed, embodied in intersubjectively useful form and put to work in coordination with other capital – because this process is a learning process, and because we show signs of learning how to learn better,<sup>76</sup> in the development of the capital structure there is a tendency to exponential growth.<sup>77</sup>

#### **3.1. Recursion**

One factor which seems to point in the direction of exponential growth is analogous to what computer scientists call recursion. Recursion is a function's making use of itself, in a kind of a feedback, or perhaps more aptly, feed-forward process. This kind of feed-forward is commonplace in capital structure development. Consider

---

<sup>76</sup> We even seem to be making progress in learning how to learn how to learn. See the work of Doug Englebart on augmentation of knowledge (1963).

<sup>77</sup> Arguably some parts of the world are experiencing exponential growth even now. In the long perspective of human history, certainly the pace of change seems to be accelerating. If we do not see present growth as exponential, perhaps that is because we are still so far down on the curve that it still looks flat.

that better steel makes possible better steel mills and better rails for transporting steel. With software, the recursion seems to be rapid and powerful.

There is, for example, a strong feed-forward dynamic between software and computer hardware. The design and manufacture of computer hardware is, of course, a demanding, complex matter. It is accordingly almost entirely computerized – under software control. But better computer hardware makes possible better computer software, in a never-ending loop. Texas Instruments is currently finishing work on the latest generation computer integrated manufacturing (CIM) system. One of the decisions they made in choosing the programming language in which to build this system was to ignore hardware requirements – it could gobble as much memory and processing power as needed; no functionality was to be sacrificed on that score. Despite the fact that Texas Instruments manufactures hardware, this sort of decision could not have been made too many years ago: processing power was too expensive. But hardware costs have dropped. Given a free hand with system size, the Texas Instruments engineers were free to choose the best available software development system, with which to build the best possible software. They chose Smalltalk and a number of related tools for the Smalltalk environment, and have purportedly produced therewith a really remarkable computer integrated manufacturing system. It is supposed to improve throughput by a factor of 100 over current methods. But notably, this system will be primarily used to produce ... computer hardware. This better, cheaper hardware may of course be used in the future to enable still more ambitious software systems ... and so the loop may continue.

Another feed-forward loop we have touched on concerns the cycle from better software construction techniques through components to markets and back to software construction techniques. The general availability, through component markets, of a wide and increasing variety of reusable components is sure to spawn new kinds of software development firms and improved software development technique, completing the loop and probably initiating some further development equally significant.

### 3.2. General computerization

In general, the effects of computerization are very significant. The benefits of computation are being extended into virtually every area of human endeavor, making possible great precision, capture of information, widespread, inexpensive communication, and a host of tools and processes that were impossible before. Consider again, for example, the tools for rapid prototyping of machine parts. Software is used in producing the drawings (computer-aided design), in sending the drawings electronically to the prototyping device, in directing the laser that hardens the polymer, and in precisely lowering the platform on which the model takes shape, layer by layer. As the software for these purposes is improved, tool prototyping will improve. Similarly, better software will impact the speed and quality of production of virtually every hard good we use.

### 3.3. Learning to use software

Beyond the simple improvement of current processes through computer use is the development of better processes that computers and software make possible. This latter effect of software on capital structure development will be the more profound.

Up to the present, in large part, we have used computers to automate old processes; we are just beginning to learn how to make use of the computer to do new and different things. Perhaps as significant, we are just beginning to learn how to adjust management techniques and organizational structures to complement the capabilities of computers.

We see this fact in the production of software itself: a great deal of work is being done on learning to manage the software production process better, to take advantage of reuse, to facilitate team programming, and to develop families of products rather than a stream of individual projects. In manufacturing fields computers provide immediate availability of information on a process, and the ability to generate what-if scenarios through computer simulations. These are powerful resources for management, such as to enable them to respond more quickly and intelligently to changing circumstances. Again, it will take us a while to learn how to use this information well, but when we do, we can expect still further advances.

Very significantly, we seem to be learning how to enable better, more rapid learning at a number of levels. Clearly the software development community has recognized the importance of building evolvable systems that can "learn" effectively. A similar awareness seems to be growing in management circles. Wheelwright and Clark urge management techniques that help producers learn from experience; Peter Senge has coined the term, "the learning organization." (1990) If indeed we achieve significant social learning on the topic of how to learn better, in the sense of how to improve our productive processes more rapidly, we certainly have a strong case for exponential growth. Our tools and processes, and

hence our productivity per person, can spiral upward without limit, outrunning the growth of population, at an accelerating pace.<sup>78</sup>

#### **4. Implications for growth theory**

How far the tendency to exponential economic development is checked, in practice, is an interesting question which we cannot pursue further here. For now, the important issue is: what are the determinants of rates of economic development? What forces tend to accelerate growth? What forces tend to impede it? These, it seems, are crucial issues with which the theory of economic growth should concern itself. We close now with a consideration of what this inquiry suggests about useful directions for the theory of economic growth.

##### 4.1. Checks to growth in the new growth theory

The new growth theory of Paul Romer, as we saw in Chapter 1, takes seriously some of the knowledge issues we have considered. Romer's work focuses on "knowledge as the basic form of capital" (1986, p. 1003), considers "endogenous technological change" (1990a), and finds that "growth rates can be increasing over time." (1986, p. 1002) With all this, we are in agreement.

Where we differ with Romer is in our views on what factors slow these tendencies to increasing rates of growth. In the simple models used by Romer and other

---

<sup>78</sup> For a persuasive presentation of a possible, indeed likely, technological basis for a marked upturn in the curve of economic development, see Drexler (1986).

growth theorists, models which assume perfect knowledge and allow for no capital destruction, there are no obvious factors tending to slow growth. But without some such impeding factors, the models would have indeterminate solutions, would go to infinity. This result being unacceptable to these theorists, they build into the models a variety of ad hoc assumptions which make them tractable, and result in some equilibrium growth path, or at least bounds on the possible rate of growth. Arrow limits the model in his learning-by-doing paper, for example, by assuming, in Romer's terms, "that the marginal product of capital is diminishing given a fixed supply of labor." (Romer 1986, p. 1006) Others rely arbitrarily on upper bounds to the production function. (Romer 1986, p. 1007) Romer himself relies, in his 1986 paper, on "diminishing returns in the research technology" (p. 1006), and in his 1990 paper on the assumption that human capital "must ultimately approach an upper bound," given fixed population. (p. S80)

In examining software development, we have found all of these restrictions to be contradicted by experience. As capital is divided and improved, its marginal product increases;<sup>79</sup> the production function – the structure of production – improves as knowledge grows and is embodied in new capital goods.

Romer's restraints seem equally insupportable by experience. He separates the "research technology" by which new designs are created from production technology and asserts that research technology is subject to diminishing returns,

---

<sup>79</sup> In this context, the idea of marginal product is metaphorical at best. The concept of marginal product is relevant where we have additional increments of the same kind of good. In this context, the essential point is that we continually have different goods, adapted to the new, more productive environment.



growth theorists, models which assume perfect knowledge and allow for no capital destruction, there are no obvious factors tending to slow growth. But without some such impeding factors, the models would have indeterminate solutions, would go to infinity. This result being unacceptable to these theorists, they build into the models a variety of ad hoc assumptions which make them tractable, and result in some equilibrium growth path, or at least bounds on the possible rate of growth. Arrow limits the model in his learning-by-doing paper, for example, by assuming, in Romer's terms, "that the marginal product of capital is diminishing given a fixed supply of labor." (Romer 1986, p. 1006) Others rely arbitrarily on upper bounds to the production function. (Romer 1986, p. 1007) Romer himself relies, in his 1986 paper, on "diminishing returns in the research technology" (p. 1006), and in his 1990 paper on the assumption that human capital "must ultimately approach an upper bound," given fixed population. (p. S80)

In examining software development, we have found all of these restrictions to be contradicted by experience. As capital is divided and improved, its marginal product increases;<sup>79</sup> the production function – the structure of production – improves as knowledge grows and is embodied in new capital goods.

Romer's restraints seem equally insupportable by experience. He separates the "research technology" by which new designs are created from production technology and asserts that research technology is subject to diminishing returns,

---

<sup>79</sup> In this context, the idea of marginal product is metaphorical at best. The concept of marginal product is relevant where we have additional increments of the same kind of good. In this context, the essential point is that we continually have different goods, adapted to the new, more productive environment.

#### 4.2. The check to growth evident here: the challenge of social learning

The present investigation suggests that in searching for factors which limit the rate of economic development, we must look elsewhere than to the kinds of limitations modeled in growth theory. In simple terms, we have found that what checks the tendency to ever more rapid economic development is that learning is challenging and time consuming. If there were perfect information, if learning were easy, and if new knowledge could be costlessly embodied in new capital goods, then growth rates would be infinite. But in fact the learning process on which economic development depends is costly in time and effort, because it is iterative and dialogical. Furthermore, much of our existing wealth has been designed in a non-modular way which makes it difficult to evolve. The learning which occurs is distributed widely throughout the capital structure in various people and tools; it does not occur in every part of that structure at once, and it takes time for relevant knowledge to spread (i.e., be learned by others, or sold to others embodied in capital goods). Social learning is also coevolutionary: it involves complex complementarities that shift in time.

As a result of these characteristics, economic development is frequently capital destroying: new knowledge often makes old obsolete. Hence the process is not cumulative; new learning and new capital cannot always be added to old.

---

This is of profound significance in the social field. We made constant use of formulas, symbols, and rules whose meaning we do not understand and through the use of which we avail ourselves of the assistance of knowledge which individually we do not possess. (1945, p. 88)

Economic development is indeterminate and path dependent: there is no equilibrium toward which it tends; there are myriad possible paths along which it can proceed. Accordingly, the process is uncertain: it requires constant readjustment of plans, constant new learning, new efforts to establish or maintain a useful place in the structure of production.

What checks economic growth rates, what restrains economic development from the unrestrained advance toward which it tends, is that learning is difficult, uncertain, and time-consuming. At present, at least, we do not seem to be very good at it. There seem to be no inherent obstacles to exponential growth; it is simply difficult to achieve. There is no fundamental tendency to diminishing marginal utility of capital, for example, nor fundamental limits to the value of the human capital we can develop in society; it is simply that for the myriad different elements of an unfathomably complex structure of production to coevolve rapidly, while maintaining a high degree of complementarity, is difficult.

#### 4.3. The learning tasks before us

We conclude where we began, with Carl Menger's assertion that

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)

Economic development depends on how well we learn. This suggests two sets of tasks, one for the practitioners of the world: the programmers, engineers, managers,

and entrepreneurs who shape the tools and processes we use in production; and one for the theorists, who try to help us understand how best to shape those tools and processes.

For the practitioners, the task is to learn how to improve the rate of social learning. Methodologies and tools must be developed which improve the dialogue through which we learn: better prototyping tools, team learning techniques, and representation schemes for facilitating communication among those with different kinds of knowledge. Especially where our systems are very complex, tools for understanding need to be developed, which offer a variety of views of the complex reality. As much as possible these representation schemes and tools for understanding should allow us to work on our complex systems in terms necessary for thinking effectively about them.

Because it is the tool systems we use in which the learning must be embodied, we need to learn better how to build more evolvable systems, systems that can "learn" effectively in an uncertain and changing world. Because modularity evidently is very important to evolvability, we need to extend our understanding of the principles of modularity and the tradeoffs among them, so as to construct systems with an appropriate degree and kind of modularity.

In the field of software development we need to learn how to achieve effective markets for software capital – software components – so that we may take advantage of the knowledge-generation that markets provide. Most important to this end, we must learn how to establish property rights to software so that we can take advantage of, rather than be hindered by, our ability to copy software almost costlessly, and to distribute it at light speed.

For theorists, the task is to understand better and explain clearly those factors which facilitate and those which impede the social learning process. These are the crucial variables that determine rates of growth. The business of growth theory should be to investigate energetically the factors which influence "the degree of progress of human knowledge," for these will determine "the degree of economic progress of mankind."

## **Bibliography**

## 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.
- Arrow, Kenneth. 1962. "The Economic Implications of Learning by Doing," Review of Economic Studies, June.
- Baetjer, Howard, and Tulloh, William. 1992. "Evolving Markets for Software Components," Hotline on Object-Oriented Technology, Vol. 4, no. 1, November.
- 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.
- Belzer, A. and Rosenfeld, L. 1987. Breaking Through the Complexity Barrier. Cambridge, Massachusetts: ICAD Publications.
- Bennet, C.H. 1985. "Fundamental Physical Limits of Computation," Scientific American, July.
- Bohm-Bawerk, Eugen von. 1959 [1889]. Capital and Interest, 3 vols. Trans. G.D. Huncke and H.F. Sennholz. South Holland, Illinois: Libertarian Press.
- Bronowski, J. 1973. The Ascent of Man, Boston: Little, Brown, & Co.
- Bosworth, George. 1992. "Objects, not classes, are the issue" Object Magazine, Vol. 2, no. 4, Nov./Dec.

- Brooks, Frederick P., Jr. 1975. **The Mythical Man-month: Essays on Software Engineering**, Reading, MA: Addison-Wesley.
- Brooks, Frederick P., Jr. 1987. "No Silver Bullet: Essence vs. Accidents of Software Engineering" **Computer**, 10-19, April.
- Chaudry, Anil. 1992. "From art to part," **Computerworld**, Vol. 26, no. 45, November 9.
- Cinquegrana, D. 1990. **Understanding ICAD System**. Cambridge, Massachusetts: ICAD Publications.
- Cox, Brad. 1990. "Planning the Software Industrial Revolution" **IEEE Software**, 25-33, November.
- Cox, Brad. 1992. **Object Technologies: A Revolutionary Approach**
- Diamond, Peter. 1990. **Growth/Productivity/Unemployment**, Cambridge, Massachusetts: MIT Press.
- Dixit, Avinash. 1990. "Growth Theory After Thirty Years," in Diamond, Peter, **Growth/Productivity/Unemployment**, Cambridge, Massachusetts: MIT Press.
- Domar, E. 1946. "Capital Expansion, Rate of Growth, and Employment," **Econometrica**, Vol. 14, 137-47.
- Domar, E. 1957. **Essays in the Theory of Economic Growth**, New York: Oxford University Press.
- Drexler, K. Eric. 1986. **Engines of Creation**. New York: Doubleday.
- Drexler, K. Eric. 1991. "Exploring Future Technologies," in J. Brockman, ed., **Doing Science**, New York: Prentice Hall.
- Englebart, Douglas C. 1963. "A Conceptual Framework for the Augmentation of Man's Intellect," in Howerton and Weeks, eds., **Vistas in Information Handling**, Washington D.C.: Spartan Books
- Gadamer, Hans-Georg. 1975. **Philosophical Hermeneutics**, trans. and ed. David E. Linge, Berkeley: University of California Press
- Goldberg, Adele. 1981. "Introducing the Smalltalk-80 System," **Byte**, Vol. 6, no. 8, August.



- Hamming, R. W. 1968. "One Man's View of Computer Science," in R.L. Ashenurst, & Susan Graham, eds., **ACM Turing Awards Lectures: The First Twenty Years 1966-1985** Reading, MA: Addison-Wesley, 1987.
- Harrod, R.F. 1939. "An Essay in Dynamic Theory," **Economic Journal**, Vol. 49, 14-33.
- 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. 1941. **The Pure Theory of Capital**, Chicago: University of Chicago Press.
- Hayek, F.A. 1945. "The Use of Knowledge in Society," in Hayek (1948).
- Hayek, F.A. 1978. **New Studies in Philosophy, Politics, Economics and the History of Ideas**, Chicago: University of Chicago Press.
- Hayek, F.A. 1979. **The Counter-Revolution of Science**, Indianapolis: LibertyPress.
- Hayek, F.A. 1988. **The Fatal Conceit**. Chicago: University of Chicago Press.
- Karrm, Lawrence J. 1990. **Designing Cost-Efficient Mechanisms: Minimum Constraint Design, Designing with Commercial Components, and Topics in Design Engineering**. New York: McGraw Hill.
- Kara, Daniel A. 1992. "CASE and Advanced Software Development on the Macintosh," **CASE Trends** Vol. 4, no. 6, September.
- Kay, Alan. 1992. "The natural history of objects," in **Happy 25th Anniversary Objects!**, Published by and supplement to SIGS Publications.
- Kirzner, Israel M. 1966. **An Essay on Capital**. New York: Augustus M. Kelley.
- Knight, Frank H. 1971 [1921]. **Risk, Uncertainty, and Profit**. Chicago: University of Chicago Press.
- KnowledgeWare. 1992. Advertisement insert, **Computerworld**, Vol. XXVI, no. 44, November.

- 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. 1978. **Capital and its Structure**. Kansas City: Sheed Andrews and McMeel.
- Lachmann, L. M. 1986. **The Market as an Economic Process**. New York: Basil Blackwell.
- Langlois, Richard 1990. "Creating External Capabilities: Innovation and Vertical Disintegration in the Microcomputer Industry." **Business and Economic History**, Second Series, Vol. 19: 93-102.
- Langlois, Richard N., and Robertson, Paul L. 1991. "Networks and Innovations in a Modular System: Lessons from the Microcomputer and Stereo Component Industries," unpublished manuscript.
- Lavoie, Don. 1985. **National Economic Planning: What is Left?**. Cambridge, Massachusetts: Ballinger.
- Lavoie, Don Baetjer, Howard , and Tulloh, William 1991a. "Coping with Complexity: OOPS and the Economist's Critique of Central Planning," **Hotline on Object-Oriented Technology**, 3: 1, (Nov) pp 6-8.
- Lavoie, Don, Baetjer, Howard and Tulloh, William. 1991b. "Increased Productivity Through Reuse: An Economist's Perspective" **Proceedings of the Third Annual Workshop on Reuse**, Software Productivity Consortium, Herndon, VA.
- Lavoie, Don, Baetjer, Howard and Tulloh, William. (1992). **The Coming Software Components Revolution: A Market-Process Perspective**.
- Leijonhufvud, Axel. 1989. "Information costs and the division of labour." **International Social Science Journal**, 120.
- Lucas, Robert E., Jr. 1988. "On the Mechanics of Economic Development," **Journal of Monetary Economics**, Vol. 22.
- Mackaay, Ejan. 1990. "Economic Incentives in Markets for Information and Innovation," **Harvard Journal of Law and Public Policy**, Summer.
- McClure, Carma. 1989. **CASE is Software Automation**, Englewood Cliffs: Prentice-Hall.

- 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.
- Mises, Ludwig von. 1966 [1949]. **Human Action**, Chicago: Henry Regnery Company.
- Mori, Ryoichi and Kawahara, Masaji. 1990. "Superdistribution: The Concept and The Architecture" **The Transactions of the IEICE**, 73:7, July.
- Mullin, Mark. 1990. **Rapid Prototyping for Object-Oriented Systems**, Menlo Park, California: Addison-Welsey.
- Nelson, Richard R., and Sidney G. Winter. 1982. **An Evolutionary Theory of Economic Change**, Cambridge: Harvard University Press.
- Norman, Ronald J., and Forte, Gene. 1992a. "Automating the Software Development Process: CASE in the '90's." **Communications of the ACM**, Vol. 35, no. 4.
- Norman, Ronald J., and Forte, Gene. 1992b. "A Self-Assessment by the Software Engineering Community." **Communications of the ACM**, Vol. 35, no. 4.
- Palmer, Tom G. 1989. "Intellectual Property: A Non-Posnerian Law and Economics Approach," **Hamline Law Review**, Spring.
- Polanyi, Michael. 1958. **Personal Knowledge**, Chicago: University of Chicago Press.
- Prieto-Diaz, Ruben. 1991. "Reuse in the U.S.," **International Conference on Software Engineering**.

- Robertson, Paul L., and Richard N. Langlois. 1992. "Modularity, Innovation, and the Firm: the Case of Audio Components," in Mark Perlman, ed., **Entrepreneurship, Technological Innovation, and Economic Growth: International Perspectives** Ann Arbor: University of Michigan Press.
- Robinson, Keith. 1992. "Putting the SE into CASE," in Spurr, Kathy, and Layzell, Paul, eds., **CASE, Current Practice, Future Prospects**, Chichester: Wiley.
- Romer, Paul M. 1986. "Increasing Returns and Long-Run Growth," **Journal of Political Economy**, Vol. 94, no. 5.
- Romer, Paul M. 1990a. "Endogenous Technological Change." **American Economic Review**, Vol. 80, no. 2.
- Romer, Paul M. 1990b. "Are Nonconvexities Important for Understanding Growth" **Journal of Political Economy**, Vol. 98, no. 5.
- 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.
- Schumpeter, Joseph A. 1934. **The Theory of Economic Development**. Cambridge: Harvard University Press.
- Senge, Peter M. 1990. **The Fifth Discipline: The Art and Practice of the Learning Organization** New York: Doubleday.
- Set Laboratories. 1992. Advertisement in **CASE Trends**, Vol. 4, no. 6, September.
- Shina, Sammy. 1991. **Concurrent Engineering and Design for Manufacture of Electronics Products**. New York: Van Nostrand Reinhold.
- 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.
- Solow, Robert M. 1956. "A Contribution to the Theory of Economic Growth," **Quarterly Journal of Economics**, Vol. 70, no. 1.
- Solow, Robert M. 1970. **Growth Theory: an Exposition**, Oxford, Clarendon.
- Sowell, Thomas. 1980. **Knowledge and Decisions**. New York: Basic Books.

- Stiglitz, Joseph E. 1990. "Comments: Some Retrospective Views on Growth Theory," in Diamond, Peter. **Growth/Productivity/Unemployment**, Cambridge, Massachusetts: MIT Press.
- Suh Nam. 1990. **The Principles of Design**. New York: Oxford University Press.
- Taylor, David A. 1990. **Object-Oriented Technology: A Manager's Guide**, Alameda, California: Servio Cororation
- Teece, David J. 1980. "Economies of Scope and the Scope of the Enterprise," **Journal of Economic Behavior and Organization**, Vol.1, no. 3.
- Tirso, Jesus. 1991. "IBM Reuse Program," **Proceedings of the Fourth Annual Workshop on Software Reuse**, Reston, Virginia, Nov. 18-22, 1991.
- Vaughn, Karen. 1990. "The Mengerian Roots of the Austrian Revival," **History of Political Economy**, supplemental issue
- 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.
- Young, Allyn. 1928. "Increasing Returns and Economic Progress." **Economic Journal**, December.

## Vita

Howard Baetjer Jr. was born on February 27, 1952 in Baltimore, Maryland. He graduated from Gilman School in 1970. He received his Bachelor of Arts in psychology from Princeton University in 1974, and then taught English for several years at St. George's School in Newport, Rhode Island. He received his Master of Letters in English literature from the University of Edinburgh in 1980, and his Master of Arts in political science from Boston College in 1984. After working for three years as educational liaison at the Foundation for Economic Education in Irvington, New York, he began doctoral work in economics at the Center for the Study of Market Processes at George Mason University. He received his Doctor of Philosophy in economics in 1993.

## PUBLICATIONS

- 1983. "Lasers, Harobeds, and World Hunger," **The Freeman**, August.
  - 1984. "Does Welfare Reduce Poverty?" **The Freeman**, April.
  - 1984. "Of Obligation and Transfer Taxation," **The Freeman**, November.
  - 1985. "Profit-Maker, Friend or Foe?" **The Freeman**, April.
  - 1986. "Deregulate the Utilities," **The Freeman**, September.
  - 1986. "Private Schools in the Inner City," **The Freeman**, November.
  - 1987. "Freedom in the Dock," **The Freeman**, February.
  - 1988. "Ebenezer Scrooge and the Free Society," **The Freeman**, December.
  - 1988. "Beauty and the Beast," **Reason**, January.
  - 1993. "The Manners of the Market," **Religion & Liberty**, March/April.
- Baetjer, Howard, and Tulloh, William. 1992. "Evolving Markets for Software Components," **Hotline on Object-Oriented Technology**, Vol. 4, no. 1, November.
- Lavoie, Don, Baetjer, Howard, and Tulloh, William. 1990. "High Tech Hayekians: Some Possible Research Topics in the Economics of Computation," **Market Process**, 8.
- Lavoie, Don, Baetjer, Howard, and Tulloh, William. 1991. "Coping with Complexity: OOPS and the Economist's Critique of Central Planning," **Hotline on Object-Oriented Technology**, Vol. 3, no. 1, November.

- Lavoie, Don, Baetjer, Howard, and Tulloh, William. 1991. "Increased Productivity Through Reuse: An Economist's Perspective" **Proceedings of the Third Annual Workshop on Reuse**, Software Productivity Consortium, Herndon, VA.
- Lavoie, Don, Baetjer, Howard and Tulloh, William. 1993. **The Coming Software Components Revolution: A Market-Process Perspective**. Boston: Patricia Seybold Group.