

# Software Economics: An Application of Price Theory to the Development of Expert Systems

Dr. J. R. Clark, Hendrix Professor of Economics, The University of Tennessee at Martin  
Dr. Leon S. Levy, Distinguished Member of the Technical Staff, AT&T Bell Laboratories

## Abstract

*"Expert Systems" in computer software appear to offer a potential for increases in project productivity in the range of Orders of Magnitude. This paper offers a software development model illustrating not only the optimal allocation of project resources between tool making and tool application but also appropriate investment in the project and several means of dealing with project risk. The model illustrates that much more benefit is to be derived from extending the domain of applicability of expert systems than from increasing the productivity gains of those techniques. We also find that it is possible to compensate for a more risky approach by alternative scheduling.*

## Introduction

Within the last ten years, the production of knowledge intensive software as exemplified by "Expert Systems" has effected very large increases in productivity. The development of these systems appears to offer a potential for increases in application productivity in the range of orders of magnitude. Since these systems depend on the knowledge of experts who are typically not programmers, there is a need for computer programs that can help the expert to write computer programs. The expert describes the domain knowledge in a notation suitable to the domain. Concurrently, a translator program is developed to transform the expert's notation into a working program. Software development using this paradigm can be quite efficient as demonstrated by Levy and Stump (1985) and Levy (1987). Traditionally, software engineers have used software costing models which extrapolate future costs from historical data but have not yet begun to seriously apply the discipline of managerial economics to software project management. Their models produce inconsistent results and fail to deal with optimal allocation of project resources as illustrated by Mohanty (1981, pp. 103-121).

It is the intent of this paper to offer a software development model which illustrates not only the optimal allocation of project resources between tool making and tool application but also considers appropriate investment in the project and several means of dealing with project risk. The model illustrates this approach by analyzing software as a production process

not unlike the production of automobiles, for example. Initially, a prototype model is built, and "road tested". Then, the production line is tooled up, and the products are turned out with lower production costs according to Gwartney, Stroup, and Clark (1985). While this metaphor has the usual shortcoming of metaphors, it does suggest a means of analysis amenable to the tools of managerial economics. The model and its analysis provides insight into the problems of software productivity and the need for a comprehensive approach rather than piecemeal attacks on parts of the problem.

## The Problem

"Expert systems" are now very large systems dealing with complex problems. In order to build and maintain such large systems, we need both a new technology and an understanding of that technology according to Levy (1987), Brooks (1975), and Balzer, Cheatham, and Green (1983, pp. 39-45). The problem can be seen in its starkest form when software productivity is contrasted with the increases in performance for a given cost (the so-called cost-performance figure of merit) that have been achieved in hardware. Conservative estimates of computer (hardware) performance show an increase in cost-performance of a factor of 2 every two years, a rate that has been unabated for almost two decades and appears to be accelerating. To maintain a constant ratio of hardware-to- software costs, software productivity should increase at a comparable rate. The required

increase in software productivity over the next decade would be a factor of at least 30.

Typical industry figures for the production of software range from 2,000 to 5,000 lines of code per staff-year. Scaling this up by the factor of 30 would require programmer productivities in the range of the equivalent of 100,000 lines of code. (Note that, in fact, the problem is considerably more severe since the complexity of programs increases superlinearly.) Thus, the increasing of software productivity to achieve a rate comparable to hardware productivity requires what we call ultra-high-programmer productivity (UHPP) techniques.

The root cause of the lagging software productivity is that software development is a labor intensive activity. Indeed, the software industry has often been described as a cottage industry. (The relative increases in software costs can thus be seen as analogous to rising costs in university tuitions and medical care, both of which are also labor intensive.) The most promising approach to achieving UHPP is through the mechanization of the software development process. The feasibility of such an approach has been documented in many places, e.g., Levy and Stump (1985) and Levy (1987). It has also given rise to a segment of the software industry that develops CASE tools for Computer Assisted Software Engineering.

However, using automated tools to increase productivity is not, in itself, sufficient. What is required is a major paradigm shift so that the whole of software development be viewed as a production process and subjected to the discipline of managerial economics to optimize the allocation and use of the factors of production. Software economics should not be merely the extrapolation of previous cost - a very inconsistent predictor as illustrated in Mohanty (1981, pp. 103-121) - but a means of assessing the most efficient way of producing software.

**The Model**

We assume that some ultra-highly productive software methodology, m, is used which is applicable to some fraction, k, of the overall software to be developed. Where this methodology is applicable, it reduces the overall software costs by a factor, g. The remaining software, 1-k, is developed as usual. P, the overall productivity gain from the use of m, is defined as:

$$EQ1.1 \quad P = E1 / E2$$

where E1 is the effort required to produce software without m, and E2 is the effort required to produce software using m. Since we are interested in only the ratio of E1/E2, we can normalize and set E1 = 1, then:

$$EQ1.2 \quad E2 = (k/g) + 1-k$$

Since P is a function of k and g then:

$$EQ1.3 \quad P(k, g) = 1/((k/g)+1-k) = g/(k+(1-k)g)$$

we are immediately interested in two cases:

$$EQ1.4a \quad \lim_{k \rightarrow 1} P \Rightarrow g$$

so that as m becomes applicable to all the software, the value of P is asymptotic to g

$$EQ1.4b \quad \lim_{g \rightarrow \infty} P \Rightarrow 1/1-k$$

so that as g becomes unbounded, P approaches a constant. Very high productivities, g, applied even to large portions of the software can be seen from equation (1.4b) to have apparently moderate effects on overall productivity (P). For example, if k = 2/3 and g = infinity, which is the case of using unlimited productivity gains on 2/3 of the software, the overall productivity increases by a factor of only 3.

**Model Dynamics**

Consider changes in overall productivity arising from changes in g and k.

$$EQ1.5 \quad dP = ((dP/dk)dk) + ((dP/dg)dg)$$

Using EQ1.3 and simplifying we get:

$$EQ1.5a \quad dP/dg = (k/g^2)P^2$$

(Note to the reader: The small letter d denotes the partial derivative symbol, thus dP/dk denotes the change in overall productivity with respect to the change in the domain of applicability k.)

$$EQ1.5b \quad dP/dk = (1-(1/g))P^2$$

the ratio of EQ1.5a and EQ1.5b is

$$EQ1.5c \quad (dP/dk)/(dP/dg) = <g^2(1-(1/g))>/k$$

From this, it is easy to see that the effect of changes in k, the domain of applicability of the ultra-high productivity technique, is much greater than the effect of changes in g (i.e., the cost reduction factor). Equation 1.5 illustrates that much more benefit is to be derived from extending the domain of applicability of expert systems than from increasing the productivity gains of those techniques.

The Effects of Doubling the Cost Savings Factor g: If we let Pn denote the increase in productivity obtained

from increasing  $g$  by a factor of  $n$  while we hold  $K$  constant: then, if  $n=2$ :

$$EQ1.6 \quad P_{,2} = (P_{,k,2g}) / (P_{,k,g}) = (1 - k + (k/g)) / (1 - k + (k/2g))$$

or

$$EQ1.6' \quad 1 + (k / (2(1-k)g)) + 0(k/g^2)$$

from this we can see that doubling  $g$  has relatively small effects on productivity. If we increase  $g$  by a factor of 2 while we hold  $k$  constant, productivity increases by 1.091.

**The Effect of Extending the Domain of Applicability:** We can recall that  $k$  is the portion of the total software to be developed to which an ultra- highly productive software methodology ( $m$ ) can be applied. Here, we examine the relative gains to be enjoyed by extending the domain of applicability for our methodology as compared to simply increasing the productivity of our methodology, itself. If we let  $r = 1-k$  represent the portion of the total software to be developed by conventional programmer productivities, then:

$$EQ1.7 \quad p(r,g) = 1 / \langle (1-r) / (g) + r \rangle$$

If we let  $P_m$  denote the change in productivity obtained from reducing  $r$  by a factor of  $1/m$  while holding  $g$  constant, we find for example:

$$EQ1.8 \quad P_{,5} = \langle (1-r) / (g) + r \rangle / \langle (1 - (r/2)) / (g) + (r/2) \rangle$$

or

$$EQ1.8' \quad P_{,5} = 2 * \{ 1 - \langle 1 / (2+r*(g-1)) \rangle \}$$

In other words, extending the domain of applicability of  $m$ , using the same relative values as we did in our example illustrating the effects of increasing  $g$  above, we find that productivity increases by a factor of 1.60. From the two above examples, it is easy to see that increasing the domain of applicability of our methodology, yields much greater increases in productivity than simply increasing the productivity of our methodology itself, i.e., gains of 1.091 for the former as compared to gains of 1.69 for the latter.

**Investing in the Project:** We assume that during the development phase, a total of  $n$  units of resources are to be expended on the project. Of these,  $n_1$  are to be initially invested in software tools and techniques for increasing productivity. The remaining  $n-n_1$  units will be used to directly produce the object code to be delivered by the project. Let  $f(n_1)$  be the rate of production of object code. Then  $f(n_1)$  can be expected to be a monotonic increasing function of  $n_1$ , since additional investment in productivity tools should never

decrease productivity.

We can therefore determine  $n_1$  which will maximize the total object code for a given fixed project budget,  $n$ . The total object code  $C$ , is given by

$$EQ1.9 \quad C = (n-n_1) f(n_1)$$

$$EQ1.9a \quad C = n f(n_1) - n_1 f(n_1)$$

To maximize  $C$ , we differentiate  $C$  with respect to  $n_1$  and set the derivative equal to 0.

$$EQ1.10 \quad dC/dn_1 = (d/dn_1) (n f(n_1) - n_1 f(n_1))$$

Then, if  $df/dn_1$  is defined as  $f'(n_1)$ , we have

$$EQ1.11 \quad n - n_1 = f(n_1) / f'(n_1)$$

$$EQ1.11a \quad n_1 = n - (f(n_1)) / (f'(n_1))$$

$$EQ1.12 \quad n_1/n = 1 - \langle (f(n_1)) / (n f'(n_1)) \rangle$$

**Linear Growth in Productivity:** We can trace out the effects of growth in productivity for linear, quadratic, and exponential rates. Linear growth in productivity can be represented by:

$$EQ1.13 \quad f(n_1) = f_0 + k n_1$$

where  $f_0$  is the initial lines of code/staff year,  $n$  is staff years, and  $k$  is lines of code/staff year/year. If the number of staff years of effort,  $N$  needed to produce the tools and techniques to double productivity is known, then EQ1.13 can be rewritten as:

$$EQ1.14 \quad 2f_0 = f_0 + kN, \text{ and then the constant } k \text{ is determined by } k = f_0/N.$$

Maximum output is obtained by choosing:

$$EQ1.15 \quad n_1 = \max(0, \langle (n/2) - (f_0/2k) \rangle)$$

Where  $n_1 > 0$ , we have:

$$EQ1.16 \quad n_1/n = \langle (1/2) - (f_0/2kn) \rangle$$

Here, we see that somewhat less than 50% of project resources should be expended to increase productivity to maximize total project output. Note also that, in the limit, when  $n$  is large, the ratio to be spent on increasing productivity should approach 50% independent of  $k$ .<sup>1</sup>

**Quadratic Growth in Productivity:** We can illustrate the effects of quadratic growth in productivity by the equation:

$$\text{EQ1.17} \quad f(n_1) = f_0 + (kn_1^2)$$

we maximize output where:

$$\text{EQ1.18} \quad n_1 = ((2n/3) - (f_0/3kn_1))$$

so that as  $n$  grows larger,  $n_1$  approaches  $2n/3$ , again independent of  $k$ .

**Exponential Growth in Productivity:** We can illustrate the effects of exponential growth in productivity by the equation:

$$\text{EQ1.19} \quad f(n_1) = f_0e^{rn_1}$$

If we differentiate EQ1.19 above, we find that

$$\text{EQ1.19a} \quad (df(n_1)/dn_1 = f_0re^{rn_1} = rf(n_1))$$

Using the basic equation to maximize output, we see that

$$\text{EQ1.20} \quad n_1 = n - (1/r)$$

Remember that the parameter  $r$  has the dimension of staff years<sup>\*</sup> - 1, and that  $1/r$  is the number of staff years to double productivity divided by the natural logarithm of 2. Also, that almost all of that time is spent on productivity enhancement, except for the interval  $1/r$  when the actual software product is generated.

### Risk

Thus far, we have considered a tool-making phase followed by a tool-application phase in production and partitioned project resources accordingly. In this two-phase method, tool application requires that tool-making be completed in sufficient time to allow for tool application and production of final product. It would be less than satisfactory on project completion date to notify a major software buyer that although the compiler program they ordered is not complete, a compiler - compiler now exists. Accordingly, the two-phase-production method is an all or nothing situation which exhibits significant risk. This risk can however be managed in the following way:

Let us define a probability distribution  $R(t)$ , which denotes the probability that the project is completed satisfactorily by time  $t$ . We assume the following:

$$\text{EQ1.21a} \quad R(t) = 0 \text{ for } t < t_0$$

where  $t_0$  is the earliest possible completion date for the project and:

$$\text{EQ1.21b} \quad R(t) = 1 \text{ as } t \text{ becomes large,}$$

as  $t$  becomes large, and  $R(t)$  is monotonically increasing. In comparing two approaches to software development, we may denote the risk of project  $i$  as  $R_i$  and the earliest possible completion date as  $t_i$ . If the form of  $R(t)$  is known for the two approaches and  $t_i$  is known for one of the approaches, then it is possible to determine what value of  $t_i$  should be chosen for the other project to achieve the same level of risk at any specified point in time.

Using this analysis, the riskier the method chosen to develop a program, the earlier its scheduled completion date must be, because its risk distribution converges more slowly. The important point here is that it is possible to compensate for a more risky approach by alternative scheduling.

### Results


The effect of changes in the domain of applicability of the ultra-high productivity techniques is much greater than the effect of changes in the cost reduction factor. From the two previous examples, it is easy to see that increasing the domain of applicability of our methodology yields much greater increases in productivity than simply increasing the productivity of our methodology itself, i.e., gains of 1.091 for the former as compared to gains of 1.69 for the latter.

When considering project investment with linear growth in productivity, somewhat less than 1/2 of project resources should be expended to increase productivity in order to maximize total project output. With quadratic productivity growth, the fraction grows to 2/3, and with exponential productivity increases, the vast majority of project resources would be allocated to productivity enhancements. In addition, the riskier the method chosen to develop a program, the earlier its scheduled completion date must be, because its risk distribution converges more slowly. It is therefore possible to compensate for a more risky approach by alternative scheduling.

### Suggestions for Future Research

The metaprogramming paradigm is based on the development of a knowledge base in a domain notation concurrently with a program development system customized to that notation. Alternate approaches to the software productivity problem are predicated on the evolution of sets of domain specific modules that can be reused without reprogramming. The primary technology involved is object-oriented programming. Both approaches require a redistribution of resources during program development. However, metaprogramming is focused on the production machinery while object-oriented programming concentrates on the subassemblies. To use an analogy, the metaprogramming ap-

proach is like a fully automated factory, while the object-oriented approach concentrates on the assembly from interchangeable standard parts. It would be useful if one could characterize the relative merits of the two approaches, even though they are clearly not mutually exclusive.

The application of price theory to software development offers the promise of approximating the optimum scale of plant with the project's first iteration. Future research might well benefit from applying the generic economic tools illustrated here in other software development areas. 

**\*\*\*Footnote\*\*\***

1. From EQ1.16, we see that  $k = f_0/n$  is a lower bound on productivity enhancement. Only when  $k$  is greater than  $f_0/n$  is the investment in productivity enhancement justified. As might be expected, this is inversely proportional to  $n$ , the project duration. Longer projects give greater opportunity to amortize investment.

**\*\*\*References\*\*\***

1. Balzer, R., T. E. Cheatham, Jr. and C. Green, "Software Technology in the 1990's: Using a New Paradigm," *IEEE Computer*, Vol. 16, No. 11, pp. 39-45, November, 1983.
2. Brooks, F. P., Jr., *The Mythical Man-month*, Addison Wesley, Workingham, UK, 1975.
3. Gwartney, J. D., R. Stroup, and J. R. Clark, *Essentials of Economics*, Academic Press, New York, NY, USA, 1985.
4. Levy, L. S., *Taming the Tiger: Software Engineering and Software Economics*, Springer-Verlag, Berlin, FRG, 1987.
5. Levy, L. S. and H. T. Stump, "Inverted Decision Tables and Their Application: Automating the Translation of Specifications to Programs," *Bell Laboratories Technical Journal*, February, 1985.
6. Mohanty, Siba N., "Software Cost Estimation: Present and Future," *Software Practice and Experience*, Vol. 11, pp. 103-121, 1981.