A Cost Analysis of the Software Dilemma: To Maintain or to Replace

Anitesh Barua **Tridas Mukhopadhyay**

Graduate School of Industrial Administration
Carnegie-Mellon University
Pittsburgh, PA 15213

Abstract

Although software maintenance claims a significant part of the data processing budget, very few authors have examined the tradeoffs between software maintenance and software replacement. We hypothesize that due to frequent modifications and functional enhancements, the system complexity increases rapidly, leading to a sharp increase in maintenance cost. Thus, there may exist a time when it is optimal to rewrite the system completely, which results in a reduction of complexity and subsequent maintenance cost. In this paper, we develop an analytical model for determining the optimal rewriting time. We consider two rewriting strategies involving old and new (superior) technologies. Several interesting propositions with managerial implications emerge out of the analysis. These include the impacts of increasing maintenance requirements and unstructuredness of the technology upon the optimal rewriting time, the differences in replacement times for old and new technologies, and the effects of system integration requirements on replacement decisions.

1. Introduction

An important problem facing both practitioners and researchers of Information Systems (IS) today is the escalating cost of software maintenance. More often than not, maintenance cost exceeds the initial development cost, and constitutes a major part of the systems life cycle cost. Empirical studies [4, 5, 14, 26] report that systems maintenance consumes between 50 and 80% of total IS resources. According to Parikh [17], worldwide software maintenance cost exceeds $30 billion per year. These figures indicate the magnitude and importance of the software maintenance problem.

Swanson [23] classifies maintenance activities as being corrective, adaptive, or perfective in nature. Lientz, Swanson and Tompkins [14] report that as much as 75% or more of maintenance work is devoted to adaptive and perfective maintenance for functional modifications and enhancements. In this paper, we study costs due to adaptive and perfective maintenance. We hypothesize that as modifications and enhancements are implemented over time, the system complexity, which is a measure of the unstructuredness of the software, increases rapidly, leading to a sharp increase in maintenance cost. Thus there may exist a time when it is optimal (in an economic sense) to rewrite the system completely, which may result in a reduction of complexity and subsequent maintenance cost. In this research we examine the option of maintaining an existing software against rewriting the system.

The use of structured programming is often advocated for easier systems maintenance [2, 9, 22]. There is some evidence that structured programs are easier to modify and maintain [7, 8, 18]. However, it is estimated that approximately 75% of the existing software are not structured [20]. Moreover, even with structured programming, it is difficult to maintain program structure due to frequent modifications and enhancements. As a result, originally structured or not, programs are expected to become more and more difficult to maintain due to the rapidly changing user needs and business environments.

With functional enhancements over time, the size of a system and the number of control flows are expected to increase. The modularity of the system may reduce, while interactions among modules may increase due to patch-up maintenance work. The increase in system size, control flows, and inter-module interactions results in higher system complexity [1, 6, 15, 24, 25], and makes system maintenance progressively more difficult [3, 11]. When the system is rewritten completely, the resulting system is expected to be much more modular with fewer inter-module interactions. The restructuring of the system is therefore expected to lower complexity, and hence its maintenance costs.

We consider two rewriting strategies in this paper. The first strategy uses the same technology (system development language and its supporting environments), with which the original system was built, for rewriting the code. The second strategy involves switching to a new (superior) technology that is intrinsically more structured and easier to modify and maintain. The advantages of the first strategy are that it does not involve any new acquisition (hardware or software), development personnel training and costly development of interfaces for meeting system integration requirements. One or more of these advantages may be lost by switching to a new technology, though the maintenance cost may be lower. We analyze these two strategies under different user environmental conditions.

In this paper, we develop a simple deterministic model of software replacement. From this model we can examine the following issues: Is it feasible to rewrite the code in order to minimize maintenance cost? If yes, what is the optimal time to do so? Is it optimal to rewrite more than once within a specified planning horizon? Under what circumstances is it better to switch to a "superior" technology in order to minimize overall cost? Are there circumstances under which it is optimal to choose an "inferior" technology for developing a system with the knowledge that it will be replaced later by a superior technology? How are system replacement decisions altered by system integration requirements?

The rest of the paper is organized as follows. Section 2 describes the model assumptions and their justifications. It also provides a limited characterization of the impact of technology on maintenance costs. Single rewriting of code using the same technology is discussed in section 3. Section 4 formalizes the strategy of switching to a more effecient technology at some point in time. The problem of choosing a technology for systems development on the basis of total life cycle cost is discussed in section 5. Section 6 generalizes the analysis of section 1 to multiple rewritings of code. System integration issues are discussed in section 7. Sections 8 and 9 contain plans for refinements/enhancements of the model and concluding remarks.

2. Model Building Blocks

In this section we describe the basic problem setting to be analyzed and state the assumptions used in building the software replacement model.

2.1 Characteristics of the problem setting

1. The IS department has a technological planning horizon $[0,T]$. That is, the IS manager does not look beyond T for his/her current decisions. At least two justifications may be provided in favor of a finite horizon. The first justification relates to the evolving nature of Information Technology. With each new generation of computers and programming languages, spectacular gains in performance/cost ratios have been achieved. Therefore, it is unlikely that a manager would make definite plans beyond a certain time, say, 15 years, by which time the current technology would become economically inefficient compared to newer technologies. The other justification refers to the inherent uncertainties associated with the general business environment that make it difficult to formulate meaningful plans after a certain time period. In summary, the assumption of a finite planning horizon seems to have ample justification. In section 3, we discuss the effects of the assumption of an exogenous planning period.

2. If a code is rewritten at any time, then there is a small "disruption" period Δ, during which neither the old system nor the new one is operational. Note that the new system has been developed in parallel with the operation of the old system. The disruption represents a transient phase of moving over to the new system. Thus, if the old system is discontinued at τ, then the new system starts steady state operation at $\tau + \Delta$.

3. We do the cost analysis in terms of real (constant) dollars.

4. m homogeneous modification/enhancement requirements arise per unit time. These requirements are created exogenously due to changing business environments and user needs. The homogeneity assumption implies that each enhancement requires the same amount of effort at a given point in time. However, over time, the effort required for implementing enhancements increases steadily due to a "deterioration" of the software from its initial state. With continuous changes and additions to the system, there is an increase in the system size, number of inter-module interactions and control flows. As a result, software complexity, which is a measure of the lack of structuredness in the software, keeps increasing with time [1, 6, 15, 24, 25]. In fact one recent study reports an increase of 60% in system complexity due to program modifications made during a single experimental seating [11]. In addition to the increase in complexity, the loss or lack of software documentation and personnel turnover make it even more difficult to enhance the system. Thus, it is expected that over time, it takes more and more effort to maintain the system.

Our assumption of continuous maintenance work is supported by Lehman [13], who, from his studies of real-world cases, concludes that a program undergoes continual change due to functional enhancements. A similar assumption is made by Gurbaxani and Mendelson [10], who assume that a program deteriorates at a uniform rate due to enhancements. Lehman [13] also observes that during its active life, a program experiences a steady growth rate, which partially justifies our assumption of homogeneous maintenance requirements.

2.2 Software costs and related assumptions

In this paper, we consider three cost components:

a) development (or rewriting) cost

b) maintenance (modification/enhancement) cost over $[0,T]$

c) disruption cost

a) Development (rewriting a system in the same or a new language) cost is approximately linearly related to the system size (SLOC, modules, functions). For example, Martin [16] and Rudolph [19] indicate that development effort is approximately linearly related to program size, both in terms of lines of code and function points.

Since m homogeneous enhancement requirements arise per unit time, the rewriting cost increases approximately linearly with time. Thus, if a system costs d to be developed at time $t = 0$, then at $t = \tau$, the system with $m\tau$ enhancements will cost $d + km\tau$ to be rewritten completely, where k is the slope of the development cost curve, and represents the incremental rewriting cost for one functional enhancement.

b) Maintenance (modification/enhancement) cost can be divided into 3 sub-components -- Analysis, coding and testing costs. For any enhancement, the software personnel first analyze the existing code to identify the portions that are going to be modified/affected due to the enhancement, and then proceed to do the necessary coding. Testing follows coding. Over time, these activities consume larger amounts of resources due to an increase in system complexity, and lead to a rapid increase in system maintenance cost.

Let c_0 be the cost of one enhancement at $t = 0$. Then the cost of m homogeneous enhancements at $t = 0$ is assumed to be mc_0. However, for all $t > 0$, the cost is more than mc_0 due to the increase in software complexity with time. Let the maintenance cost of m modifications at t be given by $m c_0 r^t$, where $r > 1$ is a constant, determined by m and the language in which the system is written. According to this cost function, the maintenance cost rises by a fixed factor r for each time period. This formulation is consistent with statements such as "maintenance costs increase by 15% each year". This regularity assumption is also supported by Lehman's finding [13] that the behavior of software evolution is mathematically "regular".

As a first cut, we assume that $r = m^\beta$, where $\beta > 0$ is a characteristic of the language in which the software is written, β being small for highly structured languages. Thus the factor r increases with the number of enhancement requirements, m, and with the unstructuredness of the technology. Consequently, the term r^t accounts for an increase in maintenance costs due to an increase in system complexity over time.

If a code is rewritten at τ, then at $\tau + \Delta$ (when the system becomes operational), the complexity is expected to diminish. Consequently the maintenance cost at $\tau + \Delta$ is also expected to reduce considerably. As a first step, we assume it to be mc_0, same as the cost at $t = 0$. In a sequel paper, we will consider the possibility that this cost may be greater than mc_0 due to the fact that the new system, however well-structured, is bigger than the initial system due to $m\tau$ enhancements.

From $t = \tau + \Delta$, the complexity of the system and hence maintenance cost start rising once again due to repeated modifications and enhancements. The maintenance cost at $t > \tau + \Delta$ is given by $mc_0 r^{t - \tau - \Delta}$.

c) A disruption (opportunity) cost is incurred during the transition from the old system to the rewritten/restructured system. This may be taken to be the first order cost of transactions that are lost or interrupted in the process. Let this cost be denoted by D.

2.3 Characterization of technology

In this paper, we use the term technology in a restricted sense. It is used to specify the software development and maintenance languages and their supporting environments. A technology will be characterized by four parameters, β, c_0, d and k, where β is a surrogate for the unstructuredness of the technology, c_0 is the cost of one modification/enhancement at $t = 0$, d is the development cost at $t = 0$, and k is the slope of the development cost curve.

Definition: A superior technology is characterized by smaller β and c_0. k, the rate at which the development cost rises, is less than or equal to the corresponding value for an inferior technology.

A superior technology is more suitable both for better restructuring of the software, and for structured programming. Thus, a superior technology should have smaller β, which is a measure of the unstructuredness of the technology. Since a superior technology is inherently better structured, the effort required for program modification/enhancement (c_0) should also be lower because it is easier to identify and recode the relevant modules with such a technology.

It is more difficult to compare the magnitude of development cost for different technologies. Some authors claim a lower development cost for superior technologies. For example, Martin [16] states that the development costs are actually less for superior technologies like fourth generation languages. Similarly, Rudolph [19] reports the results of an experiment involving a third and a fourth generation language, where the development effort ratio for the 4GL to 3GL was found to be 1:17. Thus, it seems that even the initial development cost may be lower for superior technologies. However, if new hardware has to be acquired in order to use the superior technology, then d, which now includes the initial hardware cost as well, may be higher than that for an inferior technology. But once the initial investment is made for the superior technology, the incremental cost of development (k) is not likely to exceed that for the inferior technology. Therefore, for this paper, we only assume that k should not be higher for a superior technology, without making any assertion about the development cost, d.

3. Single Rewriting of Code

In this section, we consider a situation in which the code can be rewritten only once in the interval $[0,T]$ with the original technology. However, we assume that the code is written in a structured manner within the limits of the technology. For example, the original system might have been written in unstructured COBOL, while the recreated version uses structured programming with COBOL[1]. If there exists an optimal time τ to rewrite the code, then τ may be found from the total cost function

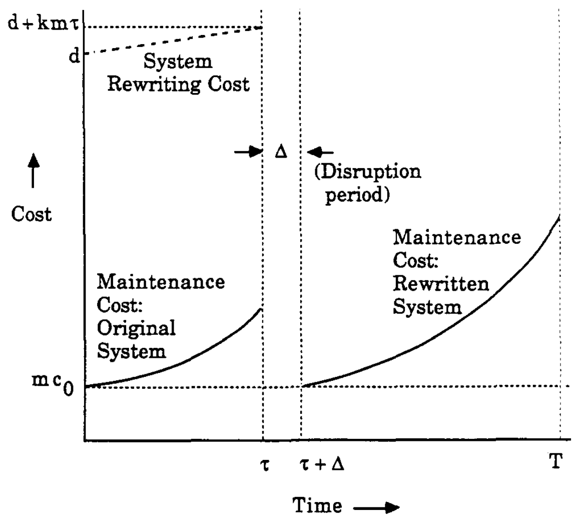
[1]Structured programming can be implemented with any language, though superior languages are intrinsically much more structured.

$$C(\tau) = \int_0^\tau mc_0 r^t dt + D + d + km\tau + \int_0^{T-\tau-\Delta} mc_0 r^t dt$$

where $\int_0^\tau mc_0 r^t dt$ and $\int_0^{T-\tau-\Delta} mc_0 r^t dt$ are the cumulative maintenance costs for the original system from $t = 0$ to τ and the rewritten system from $t = \tau + \Delta$ to T respectively, since

$$\int_{\tau+\Delta}^{T} mc_0 r^{t-\tau-\Delta} dt = \int_0^{T-\tau-\Delta} mc_0 r^t dt$$

The maintenance cost curves for the original and rewritten systems are shown in figure 1 below.



d+kmτ
d
System Rewriting Cost
Δ
(Disruption period)
Cost
Maintenance Cost: Rewritten System
Maintenance Cost: Original System
mc0
τ
τ+Δ
T
Time

Fig 1: Cost Functions for Original and Rewritten Systems

Cost of rewriting the system at τ is $d + km\tau$. Cost of m enhancements at $t = 0$ and at $t = \tau + \Delta$ is mc_0. Δ is the disruption period.

The total cost $C(\tau)$ is shown in figure 2 as a function of the rewriting time τ.

If the code is not rewritten, then the total cost is given by $\int_0^T mc_0 r^t dt$. If it is rewritten at $t = 0$, then the total cost equals $d + D + \int_0^{T-\Delta} mc_0 r^t dt$. A rewriting at T costs $d + kmT$ and the system is not put to use. Obviously, $t = 0$ and T are not candidates for rewriting times. There is, however, no guarantee for the éxistence of an optimal time to rewrite. By differentiating $C(\tau)$ and by solving for τ, we can find the minimum $C(\tau^*)$. But we must check whether $C(\tau^*) < \int_0^T mc_0 r^t dt$. If it is true, then it is optimal to rewrite at τ^*.

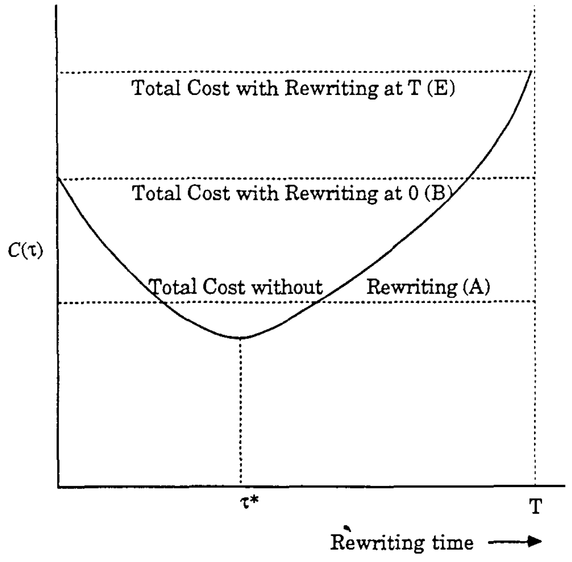


Fig 2: Total Maintenance Cost as a Function of Rewriting Time

Optimal rewriting time is τ^*. Total cost without rewriting is A $= \int_0^T mc_0 r^t dt$. Total cost with rewriting at $t = 0$ is B $= d + D + \int_0^{T-\Delta} mc_0 r^t dt$. Total cost with rewriting at T $= d + kmT + \int_0^T mc_0 r^t dt$.

Proposition 1: If it is optimal to rewrite the code at τ, then τ is less than half the technological planning horizon T.

Proof:

$$C(\tau) = \frac{mc_0}{\log r}[r^\tau - 1] + d + km\tau + D + \frac{mc_0}{\log r}[r^{T-\tau-\Delta} - 1]$$

The first order condition is given by

$$mc_0 r^\tau - mc_0 r^{T-\tau-\Delta} + km = 0$$

or $r^{T-\tau-\Delta} - r^\tau > 0$

or $\tau < \frac{T-\Delta}{2}$ which implies $\tau < T/2$. Q.E.D.

The explanation of this result is as follows: The optimal rewriting time is such that the difference of the marginal[2] maintenance costs for the original and the rewritten systems must equal the marginal cost of rewriting. As the rewriting time τ increases, the marginal cumulative cost for the original system increases, while that of the rewritten system decreases. The rate of decrease must be greater than the rate of increase so that the difference is negative. This is possible only when $\tau < \frac{T-\Delta}{2}$. In fact, if $k = 0$, then the original

[2]With respect to the rewriting time τ.

system will be discontinued exactly at $\frac{T-\Delta}{2}$. This proposition provides the manager with a simple mechanism for checking if it is too early or late for rewriting the system.

A discussion of the effects of the finite horizon $[0,T]$ is in order. From this proposition, it would seem that for planning horizons that are too small, it may not be feasible to rewrite the system. However, we can rule out such a short planning horizon. For our experience with the evolution of Information Technology shows that four generations of computer hardware and software have evolved since the early 1950's. On an average, each generation has evolved almost over a decade. This rate of technological progress, although relatively fast, does not require a very short planning horizon. At least, the planning horizon is not expected to be so short as to make rewriting infeasible.

Technological planning horizons that are too long may, however, pose problems for the single rewriting case. With large T, the original system may be operational for too long before getting rewritten. Fortunately, this shortcoming of the model is overcome in section 6, where multiple rewritings of the code are considered. With multiple rewritings, an increase in T merely increases the number of rewrites.

Proposition 2: The operational life of a system increases with an increase in modification/enhancement requirements and decreases with increased structuredness of the technology.

Proof:

Note that higher β implies less structured technology. Since $r = m^{\beta}$, $\frac{\partial r}{\partial m}$ and $\frac{\partial r}{\partial \beta} > 0$. Also $\frac{\partial \tau}{\partial m} = \frac{\partial \tau}{\partial r} \cdot \frac{\partial r}{\partial m}$ and $\frac{\partial \tau}{\partial \beta} = \frac{\partial \tau}{\partial r} \cdot \frac{\partial r}{\partial \beta}$

From the implicit function theorem, $\frac{\partial \tau}{\partial r} = -\frac{F_r}{F_\tau}$ where

$$F_\tau = c_0\, r^{\tau} \log r + c_0\, r^{T-\tau-\Delta} \log r > 0 \quad \text{and}$$

$$F_r = \frac{c_0}{r} [\tau\, r^{\tau} - (T-\tau-\Delta)\, r^{T-\tau-\Delta}] < 0 \ \text{ since } \ \tau < \frac{T-\Delta}{2}$$

$$\text{Therefore, } \frac{\partial \tau}{\partial r} > 0 \Rightarrow \frac{\partial \tau}{\partial m} > 0,\ \frac{\partial \tau}{\partial \beta} > 0. \quad \text{Q.E.D.}$$

According to this proposition, as long as $\tau < \frac{T-\Delta}{2}$, one would expect systems to be operational for a longer period of time (before a rewrite) in more dynamic business environments with more frequent needs for enhancements. Similarly, a more unstructured technology results in a longer operational life of a system.

This proposition may seem counter-intuitive at first glance. With increasing m, one would expect the system to become more complex in a shorter period of time and to warrant rewriting earlier. However, it should be noted that the code is rewritten using the same technology and that the rewritten system takes a greater share of the maintenance cost. Thus, a shortening of the operational life of the original system with an increase in m leads to a large increase in the maintenance cost of the rewritten system. Therefore, the system is actually rewritten later with an increase in m. A similar remark applies to the second part of the proposition.

It is important to note here that $\frac{\partial \tau}{\partial \beta} > 0$ only when the code is rewritten using the same technology. Later on we show that when switching to a superior technology is considered, an inferior technology is replaced earlier, i.e., $\frac{\partial \tau}{\partial \beta} < 0$.

4. Switching to Superior Technologies

As mentioned before, empirical evidence [16, 19] indicates that advanced technologies like 4GLs make programming significantly less time consuming. Sprague and McNurlin [21] emphasize that 4GLs should be used to rewrite old software for greater flexibility and maintainability. Sometimes, it may indeed be beneficial to switch to more structured technologies that are inherently easier to modify and maintain. Of course, there are switching costs in form of acquisition/development, learning and integration. In this section, we only consider the acquisition/development cost associated with a new technology. Issues related to system integration are discussed in a later section.

For a new technology, let the development/acquisition cost at time t be given by $d_n + k_n mt$. Additional hardware acquisition cost, if any, is lumped into d_n. Note that the cost increases with time because of an increase in the functional requirements.

Proposition 3: Let a current technology be characterized by β_1, c_1, d_1 and k_1. Let $r_1 = m^{\beta_1}$. Let a new (superior) technology be characterized by β_n, c_n, d_n and k_n, such that $\beta_1 > \beta_n$, $c_1 > c_n$, $k_1 = k_n$. Note that we do not specify the direction of the inequality for d_1 and d_n. Let τ_1 and τ_n be the optimal times to rewrite the code using the original and new technologies respectively. Then $\tau_n < \tau_1$.

Proof:

For rewriting using the same technology, the first order condition is given by

$$c_1 r_1^{\tau_1} - c_1 r_1^{T-\tau_1-\Delta} + k_1 = 0$$

For switching to the new technology at τ_n, we have

$$c_1 r_1^{\tau_n} - c_n r_n^{T-\tau_n-\Delta} + k_n = 0$$

Since $k_1 = k_n$,

$$c_1 [\, r_1^{\tau_n} - r_1^{\tau_1} \,] = c_n r_n^{T-\tau_n-\Delta} - c_1 r_1^{T-\tau_1-\Delta}$$

If $\tau_n > \tau_1$, then the left hand side of the above equation is greater than zero, while the right hand side is less than zero. This leads to a contradiction. Similarly, $\tau_n = \tau_1$ leads to another contradiction. Therefore, the proposition follows. Q.E.D.

This proposition shows that switching to a superior technology occurs earlier than rewriting with the same technology. The reasoning is as follows. Since, $k_1 = k_n$ by hypothesis, the difference in the marginal maintenance costs for the original and the rewritten systems must be equal for both the superior and the old technologies. However, since $c_1 r_1 > c_n r_n$, τ_n must be less than τ_1, so that the marginal cost difference may still be equal for the two technologies.

Note that the proposition does not imply that switching to a superior technology is better than rewriting the code using the old technology. The alternative actions in the calculation of τ_1 are (i) stay with the old system and do not rewrite and (ii) rewrite using the old technology. The alternatives for τ_n are (i) stay with the old system without rewriting and (ii) switch to the new technology. In order to determine the optimal action, total costs for the two alternatives must be compared. In fact the initial development/acquisition cost may be so high for the new technology that it is optimal to rewrite the system using the same technology[3]. Later on we discuss how integration requirements create another switching barrier.

Proposition 4: With switching to new technologies, inferior technologies will be replaced earlier (opposite for rewriting with the same technology).

Proof:

Consider three technologies 1, 2 and 3 characterized by $\{\beta_1, c_1, k_1, d_1\}$, $\{\beta_2, c_2, k_2, d_2\}$ and $\{\beta_n, c_n, k_n, d_n\}$ respectively. Let 1 and 2 be two current technologies and let 3 be a new one (superior to both 1 and 2). Let 2 be superior to 1, so that we have $c_1 > c_2 > c_n$, $\beta_1 > \beta_2 > \beta_n$.

If it is optimal to switch to 3 from i at τ_{in}, $i = 1,2$, then we have

$$c_i r_i^{\tau_{in}} - c_n r_n^{T-\tau_{in}-\Delta} + k_n = 0 \text{ and}$$
$$c_1 r_1^{\tau_{1n}} - c_2 r_2^{\tau_{2n}} = c_n \left[r_n^{T-\tau_{1n}-\Delta} - r_n^{T-\tau_{2n}-\Delta} \right]$$

Suppose $\tau_{1n} > \tau_{2n}$. Then this leads to a contradiction because the left and right hand sides become greater than and less than 0 respectively. Therefore, $\tau_{1n} < \tau_{2n}$, implying that the inferior technology 1 will be replaced earlier. Q.E.D.

This result is the opposite of what we obtained earlier for rewriting with the same technology. The explanation is that in this case, a superior technology with lower maintenance cost is used to replace an old technology; if the switching occurs later with increased unstructuredness of the old technology, then the maintenance cost rises even more sharply during the additional period over which the old technology must be operational. Thus the switching must take place earlier with decreasing structuredness of the old technology.

[3]Especially if new hardware has to be acquired.

5. Initial Choice of Technology

So far we have not taken into account the choice of technology for the original system. We have solely been concerned with the operation phase of the system, and have assumed that a system already exists at $t = 0$. However, the ultimate objective is to minimize system life cycle costs (development plus maintenance). The initial choice of technology can have a significant impact on the operation phase costs. Thus the initial technology related decision should be taken after a careful consideration of the active life of the system.

Let two technologies 1 and 2 be currently available. Let 1 be superior to 2 in terms of c_i and β_i, $i = 1, 2$. Let the development cost at $t = 0$ be d_i for technology i and let $d_1 > d_2$. This does not necessarily mean that it costs more to write the code using technology 1[4]. Technology 1 may involve the cost of acquiring new hardware, extensive training both for development personnel and users, etc., which can be lumped into d_1. Let $M^i_{0,T}$ denote the cumulative maintenance cost from 0 to T for technology i. If the code is to be rewritten using the same technology, then

$$M^i_{0,T} = \int_0^{\tau_i} mc_i r_i^t dt + \int_0^{T-\tau_i-\Delta} mc_i r_i^t dt + d_i + k_i m\tau_i + D$$

otherwise $M^i_{0,T} = \int_0^{T} mc_i r_i^t dt$

In absence of the knowledge of any other technology, let it be optimal to choose technology 1, i.e.,

$$d_1 + M^1_{0,T} < d_2 + M^2_{0,T}$$

However, it may not be optimal to choose the superior technology, if a new technology 3 (superior to both 1 and 2) is going to be available at t, $0 < t < T$.

Proposition 5: If a new technology is available at t, and if τ_{1n} and τ_{2n} are the optimal switching times for technologies 1 and 2 respectively, then the inferior technology 2 will be chosen if the following condition is satisfied:

$$d_1 - d_2 \geq m \int_0^{\tau_{1n}} \left[c_2 r_2^t - c_1 r_1^t \right] dt$$

The left hand side represents the difference in development/acquisition costs. The right hand side is the difference in cumulative maintenance costs for technologies 2 and 1 from $t = 0$ to τ_{1n}, the optimal time for replacing technology 1.

Proof:

With replacement by technology 3, $M^2_{0,T}$

$$= \int_0^{\tau_{2n}} mc_2 r_2^t dt + \int_0^{T-\tau_{2n}-\Delta} mc_n r_n^t dt + d_n + k_n m\tau_{2n} + D$$

[4]Earlier we referred to studies that claim it costs much less!

$$< \int_0^{\tau_{1n}} mc_2 r_2^t dt + \int_0^{T-\tau_{1n}-\Delta} mc_n r_n^t dt + k_n m\tau_{1n} + d_n + D$$

since by hypothesis, τ_{2n} is the optimal time to switch from technology 2 to 3. Therefore, the difference in life cycle costs for 1 and 2 in presence of 3 is given by

$$d_1 - d_2 + M^1_{0,T} - M^2_{0,T} >$$

$$d_1 + \int_0^{\tau_{1n}} mc_1 r_1^t dt + \int_0^{T-\tau_{1n}-\Delta} mc_n r_n^t dt +$$
$$d_n + k_n m\tau_{1n} + D - d_2 - \int_0^{\tau_{1n}} mc_2 r_2^t dt -$$
$$\int_0^{T-\tau_{1n}-\Delta} mc_n r_n^t dt - d_n - k_n m\tau_{1n} - D$$

$$= d_1 - d_2 - m\int_0^{\tau_{1n}} [\, c_2 r_2^t - c_1 r_1^t \,]\, dt$$

Obviously the right hand side of the last inequality (and hence the left hand side) is ≥ 0 if the (sufficient) condition of the proposition is satisfied. Under this condition, it is economically justified to choose 2 as the initial technology. Q.E.D.

This proposition specifies a sufficient condition under which it is optimal to choose an inferior technology for applications development with the knowledge that the system will be rewritten with a new technology at a later point in time. Note that in absence of this new technology, it is optimal to choose the currently available superior technology 1. Without the new technology, the maintenance cost difference between technologies 1 and 2 becomes prominent towards later stages. The new technology reduces this maintenance cost difference, and thus the technology with lower development cost becomes more attractive as the current choice.

For example, there are many 3GLs that are superior to COBOL both in terms of development and maintenance costs. However, the IS personnel of an organization may be highly trained in COBOL as a development technology. For developing a new application with a superior 3GL, the IS personnel may have to undergo considerable training to be able to write the code efficiently. Thus the initial training plus development cost is probably much higher for the superior technology, though it is easier to maintain and modify. The maintenance cost difference may still be sufficiently large to favor the development of the new application with the superior 3GL. If, however, the IS managers anticipate switching to a much superior technology like a 4GL sometime in the future, then the current development decisions must be reconsidered. If the condition of proposition 6 is satisfied, it is indeed optimal to keep using COBOL for all new applications until the time to switch to the 4GL.

6. Multiple Rewritings of Code

So far we have considered rewriting ~~the~~ code only once, if necessary. However, for rapidly changing environments, it may indeed be optimal to rewrite the code several times. Since it is difficult to determine simultaneously the optimal number of times the code should be rewritten and the best times for doing so, we deal with these two decision variables separately.

First, let us consider n rewritings of the code, $n > 1$. Let τ_i be the timing of the ith rewriting, $i = 1, 2, \ldots, n$.

Proposition 6: The operational life of the nth rewritten system is the longest, while that of the original system is the shortest.

Proof:

For n rewritings, the total life cycle cost is given by

$$\int_0^{\tau_1} mc_0 r^t dt + \int_0^{\tau_2-\tau_1-\Delta} mc_0 r^t dt + \ldots\ldots +$$
$$\int_0^{T-\tau_n-\Delta} mc_0 r^t dt + (n+1)d + km\tau_1 + \ldots\ldots + km\tau_n + nD$$

From the first order conditions, we have

$$mc_0\,[\, r^{\tau_2-\tau_1-\Delta} - r^{\tau_1} \,] = km$$
$$\vdots$$
$$mc_0\,[\, r^{T-\tau_n-\Delta} - r^{\tau_n-\tau_{n-1}-\Delta} \,] = km$$

Therefore, $\tau_2 - \tau_1 - \Delta$ (operational life of the first rewritten system) $> \tau_1$ (operational life of the original system) and $T - \tau_n - \Delta$ (life of the nth rewritten system) $> \tau_n - \tau_{n-1} - \Delta$ (life of the n-1th rewritten system). Q.E.D.

Single rewriting is just a special case of the the n-rewritings formulation. Proposition 6 is a generalization of the result that the operational life of the rewritten system is longer than that of the original system in the single rewriting case.

A straightforward corollary of the proposition is that if $k = 0$, i.e., if the rewriting cost does not increase with time, then each rewritten system is operational for $\frac{T-n\Delta}{n+1}$. This is possible when the rewriting cost stays relatively constant due to increasing familiarity with the technology and the system functionality, even though the size of the system increases over time.

Let us consider this special situation in more detail. In particular, let us find out the optimal number of rewritings n, when each rewritten system is operational for the same duration.

For n rewritings, the total life cycle cost is given by

$$\frac{mc_0(n+1)}{\log r}\,[\, r^{[T-n\Delta]/[n+1]} - 1 \,] + (n+1)d + nD$$

The optimal number of rewritings n^* can be found from the first order condition.

Proposition 7: The number of rewritings increases with the unstructuredness of the technology if $\frac{T+\Delta}{n+1} \log r > 1$.

Proof:

The first order condition is given by

$$\frac{mc_0}{\beta \log m}\left[m^{\beta[T-n\Delta]/[n+1]} - 1 \right] - \frac{mc_0(T+\Delta)}{n+1} m^{\beta[T-n\Delta]/[n+1]} + d + D = 0$$

$\frac{\partial n}{\partial \beta} = -\frac{F_\beta}{F_n}$ where

$$F_n = \frac{mc_0(T+\Delta)^2 \log r \;\; r^{[T-n\Delta]/[n+1]}}{(n+1)^3} > 0$$

and $F_\beta = -\frac{mc_0}{\beta^2 \log m}\left[r^{[T-n\Delta]/[n+1]} - 1 \right] - \frac{mc_0(T-n\Delta)}{n+1} r^{[T-n\Delta]/[n+1]} \left[\frac{T+\Delta}{n+1} \log m - \frac{1}{\beta} \right]$

Therefore, if $\left[\frac{T+\Delta}{n+1} \right] \log r > 1$, then $F_\beta < 0$. Q.E.D.

This proposition provides a sufficient condition under which a system developed with a more unstructured technology is rewritten more often.

7. System Integration Effects

Very often, organizations continue to maintain old inefficient systems much longer than the optimal times that would be predicted by our model. This may not necessarily constitute an irrational decision on the part of MIS managers. Switching costs almost always accompany the adoption of a new technology. Such costs may be made up of one or more of the following: Development/acquisition cost, cost of integration with existing applications and learning/training cost.

Development/acquisition cost has been considered earlier. In this section, we emphasize integration cost. Typically, applications will be sharing software modules from a large software library. We use the term "communication" generically to denote this sharing. There may be a many-to-many communication between applications and the software library modules. However, certain modules will definitely be more critical than others in terms of their calling frequencies by applications. Also, certain modules will be critical in terms of modifications requirements and hence maintenance costs.

Let us first consider the software library programs[5]. From a maintenance perspective, it is probably justified to rewrite the programs of the software library using a more efficient technology. But if it leads to problems of communication with applications which can only be remedied by a costly development of interfaces, then the replacement may not be economically feasible.

Let there be p applications sharing s programs in a software library. For each program $i, i = 1, 2, \ldots, s$, we need to compute ΔM_i, the the maintenance cost that could be saved by switching to a superior technology. This has been discussed in sections 3 and 4. If a new technology is used to rewrite one or more of the software library programs, then it may be necessary to write interfaces to allow communication between the applications and the rewritten program(s). However, it may not be necessary to write interfaces for every program that is rewritten. First, certain languages have features by which programs written in those languages can easily communicate with programs in other languages. Secondly, writing a single interface may allow communication between a subset of the software library programs and applications.

For any program which requires an interface, we need to identify all other programs that can utilize the same interface for communications. The sum of ΔM's for this group of programs has to be calculated. If the sum exceeds the cost of writing the interface, then the entire group should be rewritten with a more efficient technology. In this approach, we assume that a lack of communication between applications and software library programs is too costly and must be avoided by writing interfaces.

A similar approach may be used to determine the optimal rewriting plan for the application programs.

8. Model Refinements/Enhancements

The model that we presented is simple and exploratory in nature and does not incorporate many features of the real world. Using the current model as a starting point, our future research will follow the undermentioned plan.

1. We would like to allow for stochastic modification requirements. If this feature is considered in the model, then the rewriting times will no longer be deterministic. In that case some optimal-stopping-rule technique will have to be used to find the optimal rewriting policy.

2. In this paper, we assumed that all modification requirements are homogeneous. The model will be more generalizable if we allow modification requirements to follow some specified distribution.

[5]A program may consist of many modules.

9. Conclusions

Software budgets in organizations are growing rapidly, and this trend is expected to continue through the remaining part of the century. Empirical evidence suggests that a major fraction of the software cost is incurred through software maintenance in form of modifications/enhancements to the original system. In this paper, we attributed the increase in maintenance cost over time to an increase in software complexity, resulting from frequent changes and additions to the system. We suggested that rewriting the system at certain optimal times can lower the complexity through restructuring of the code, and thereby reduce the cost of maintaining the system.

We developed a simple deterministic model to determine optimal software rewriting/replacement times. A survey of the pertinent literature indicates the absence of any formal model in this important domain of Information Systems. This exploratory model is intended to serve as a basis for more rigorous research in this area.

With required data inputs, the model can determine whether it is optimal to i) stay with the old system ii) rewrite the code at time τ or to iii) switch to a new (superior) technology at some point in time. We generated several interesting propositions from the model. In particular, we characterized the impacts of increasing user requirements and the unstructuredness of the technology upon the optimal system replacement times. We also studied the effects of the availability of superior technologies and costs of acquiring, developing and integrating with existing systems upon the replacement decisions.

The limitation of this model arises from the form of the cost function that we used. We suggest two possible directions for making the model a practical tool. First, it is possible to gather data on the life cycle costs of a system in the form of a time series. We can either estimate the parameters of the cost function (m, r, c_0, β) from the data or determine a new cost function through regression analysis. If the latter option is chosen, then the new cost function can still be used in the same way to determine the optimal replacement times.

This paper has been an attempt at formal modelling in an area that has primarily been dominated by rules of thumb. It is hoped that the simple analysis presented in this paper will lead to more formal models of software maintenance and generate important insights into the problems and relevant issues.

Acknowledgements

We have benefited from discussions with Prof. Charles Kriebel and Dani Zweig during the problem formulation stages of this research. The helpful comments and suggestions made by Prof. Haim Mendelson and the anonymous referees are gratefully acknowledged.

References

1. Bowen, J.B., "Are Current Approaches Sufficient For Measuring Software Quality?", *Proceedings of the ACM Software Quality Assurance Workshop*, November 1978, pp. 148-155.

2. Colter, M.A., "Evolution of the Structured Methodologies", in J.D. Cougar, M.A. Colter and R.W. Knapp, *Advanced System Development/Feasibility Techniques*, 1982, New York: Wiley, pp. 73-96.

3. Curtis, Bill et al, "Measuring the Psychological Complexity of Software Maintenance Tasks with the Halstead and McCabe Metrics", *IEEE Trans. On Software Engineering*, Vol. SE-5, No. 2, March 1979, pp. 96-104.

4. Elshoff, J.L., "An Analysis of Some Commercial PL/1 Programs", *IEEE Trans. on Software Engineering*, Vol. SE-2, No. 2, June 1976.

5. Freedman, David, H., "Programming Without Tears", *High Technology*, Vol. 6, No. 4, April 1986, pp. 38-45.

6. Halstead, M.H., "Elements of Software Science", Elsevier, New York, 1977.

7. Holton, J.B., "Are the New Programming Techniques Being Used?", *Datamation*, Vol. 23, July 1977, pp. 97-103.

8. Hugo, I.S., "A Survey of Structured Programming Practice", *Proceedings of the AFIPS Conference*, New York, 1977, pp. 741-752.

9. Gane, C. and Sarson, T., "Structured Methodology: What Have We Learned?", *Computer World*, Extra, Vol 14, No. 38, September 1980, pp. 52-57.

10. Gurbaxani, Vijay and Mendelson, Haim, "Software and Hardware in Data Processing Budgets", *IEEE Transactions on Software Engineering*, Vol. SE-13, No. 9, September 1987, pp. 1010-1017.

11. Kafura, Dennis and Reddy, Geereddy, "The Use of Software Complexity Metrics in Software Maintenance", *IEEE Transactions on Software Engineering*, Vol. SE-13, No. 3, March 1987, pp. 335-343.

12. Kolodziej, Stan, "Gaining Control of Maintenance", *Computerworld Focus*, 20 (7A), February 1986, pp. 31-36.

13. Lehman, M.,"Programs, Life Cycles, and Laws of Software Evolution", in *Tutorial on Software Maintenance*, edited by Parikh, G. and Zvegintzov, N., IEEE Computer Society, LA, Calif., 1982, pp. 199-215.

14. Lientz, B.P., Swanson, E.B. and Tompkins, "Characteristics of Application Software Maintenance", *Communications of ACM*, Vol. 21, No. 6, June 1978, pp. 466-471.

15. McCabe, T.J., "A Complexity Measure", *IEEE Trans. on Software Engineering*, Vol. SE-2, No. 4, December 1976, pp. 308-320.

16. Martin, James, "Fourth Generation Languages Volume I, Principles", Prentice Hall, Inc., Englewood Cliffs, NJ, 1985.

17. Parikh, G., "Restructuring Your COBOL Programs", *Computerworld Focus*, 20 (7A), February 1986, pp. 39-42.

18. Rombach, H.D., "A Controlled Experiment on the Impact of Software Structure on Maintainability", *IEEE Trans. on Software Engineering*, Vol. SE-13, No. 3, March 1987, pp. 344-354.

19. Rudolph, E.E., "Productivity in Computer Application Development", University of Auckland, New Zealand, 1984.

20. Schneidewind, Norman, "The State of Software Maintenance", *IEEE Trans. on Software Engineering*, Vol. SE-13, No. 3, March 1987, pp. 303-310.

21. Sprague, Ralph, H. and McNurlin, Barbara, C., editors, "Information Systems Management in Practice", Prentice Hall, Englewood Cliffs, NJ, 1986.

22. Stevens, W.P., Myers, G.J., and Constantine, L.L., "Structured Design", *IBM Systems Journal*, Vol. 13, No. 2, 1974, pp. 115-139.

23. Swanson, E.B., "The Dimensions of Maintenance", in *Proceedings of the Second International Conference on Software Engineering*, 1976, pp. 492-497.

24. Woodfield, S.N., "Enhanced Effort Estimation by Extending Basic Programming Models to Include Modularity Factors", unpublished thesis, Purdue University, 1980.

25. Yao, S.S. and Collofello, J., "Some Stability Measures for Software Maintainers", *IEEE Trans. on Software Engineering*, Vol. SE-6, No. 6, November, 1980.

26. Zelkowitz, M.V., "Perspectives on Software Engineering", *Comput. Surveys*, Vol. 10, No. 2, June 1978.