THE ECONOMICS OF SOFTWARE MAINTENANCE IN THE TWENTY FIRST CENTURY

Version 3 – February 14, 2006

Abstract

All large companies utilize software in significant amounts. Some companies exceed 1,000,000 function points in the total volume of their corporate software portfolios. Much of this software is now more than 10 years old, and some applications are more than 25 years old. Maintenance of aging software tends to become more difficult year by year since updates gradually destroy the original structure of the applications.

Starting at the end of the twentieth century a series of enormous maintenance problems began to occur. The first of these problems consisted of the software updates necessary to support the unified European currency or Euro. The second problem consisted of the software updates to repair or and minimize the impact of the Year 2000 software bug in existing portfolios. Two similar problems that will occur later in the century will be the need to add digits to U.S. telephone numbers and to add digits to social security numbers.

The resources devoted to the Euro and Y2K problems caused delays in many other projects. Mass-update and other maintenance projects will potentially absorb almost 70% of the world's software professionals during much of the 21st century. Mass update software projects can top five trillion dollars in overall costs before the middle of the twenty first century. It is obvious that better maintenance tools and technologies are an urgent global priority.

Capers Jones, Chief Scientist Emeritus
Software Productivity Research, Inc.

Email CJones@SPR.com
Web http://www.spr.com

THE ECONOMICS OF SOFTWARE MAINTENANCE
IN THE TWENTY-FIRST CENTURY

INTRODUCTION

As the twenty-first century advances more than 50% of the global software population is engaged in modifying existing applications rather than writing new applications. This fact by itself should not be a surprise, because whenever an industry has more than 50 years of product experience the personnel who repair existing products tend to outnumber the personnel who build new products. For example there are more automobile mechanics in the United States who repair automobiles than there are personnel employed in building new automobiles.

At the end of the twentieth century software maintenance grew rapidly during 1997-2000 under the impact of two "mass updates" that between them are required modifications to about 85% of the world's supply of existing software applications.

The first of these mass updates was the set of changes needed to support the new unified European currency or Euro that rolled out in January of 1999. About 10% of the total volume of world software needed to be updated in support of the Euro. However in the European Monetary Union, at least 50% of the information systems required modification in support of the Euro.

The second mass-update to software applications was the "Y2K" or year 2000 problem. This widely discussed problem was caused by the use of only two digits for storing calendar dates. Thus the year 1998 would have been stored as 98. When the century ended, the use of 00 for the year 2000 would violate normal sorting rules and hence cause many software applications to fail or to produce incorrect results unless updated.

The year 2000 problem affected as many as 75% of the installed software applications operating throughout the world. Unlike the Euro, the year 2000 problem also affected some embedded computers inside physical devices such as medical instruments, telephone switching systems, oil wells, and electric generating plants.

Although these two problems were taken care of, the work required for handling them triggered delays in other kinds of software projects and hence made software backlogs larger than normal.

Under the double impact of the Euro conversion work and year 2000 repair work it is appeared that more than 65% of the world's professional software engineering population was engaged in various maintenance and enhancement activities during 1999 and 2000.

Although the Euro and the Y2K problem are behind us, they are not the only mass-update problems that we will face. For example it may be necessary to add one or more digits to

U.S. telephone numbers by about the year 2015. The UNIX calendar expires in the year 2038 and could troublesome like the year 2000 problem. Even larger, it may be necessary to add at least one digit to U.S. social security numbers by about the year 2050.

The imbalance between software development and maintenance is opening up new business opportunities for software outsourcing groups. It is also generating a significant burst of research into tools and methods for improving software maintenance performance.

What is Software Maintenance?

The word "maintenance" is surprisingly ambiguous in a software context. In normal usage it can span some 21 forms of modification to existing applications. The two most common meanings of the word maintenance include: 1) Defect repairs; 2) Enhancements or adding new features to existing software applications.

Although software enhancements and software maintenance in the sense of defect repairs are usually funded in different ways and have quite different sets of activity patterns associated with them, many companies lump these disparate software activities together for budgets and cost estimates.

The author does not recommend the practice of aggregating defect repairs and enhancements, but this practice is very common. Consider some of the basic differences between enhancements or adding new features to applications and maintenance or defect repairs as shown in table 1:

Table 1: Key Differences Between Maintenance and Enhancements

	Enhancements (New features)	**Maintenance (Defect repairs)**
Funding source	Clients	Absorbed
Requirements	Formal	None
Specifications	Formal	None
Inspections	Formal	None
User documentation	Formal	None
New function testing	Formal	None
Regression testing	Formal	Minimal

Because the general topic of "maintenance" is so complicated and includes so many different kinds of work, some companies merely lump all forms of maintenance together and use gross metrics such as the overall percentage of annual software budgets devoted to all forms of maintenance summed together.

This method is crude, but can convey useful information. Organizations which are proactive in using geriatric tools and services can spend less than 30% of their annual software budgets on various forms of maintenance, while organizations that have not used any of the geriatric tools and services can top 60% of their annual budgets on various forms of maintenance.

Although the use of the word "maintenance" as a blanket term for more than 20 kinds of update activity is not very precise, it is useful for overall studies of national software populations. Table 2 shows the estimated U.S. software population for the United States between 1950 and 2025 divided into "development" and "maintenance" segments.

In this table the term "development" implies creating brand new applications or adding major new features to existing applications. The term "maintenance" implies fixing bugs or errors, mass updates such as the Euro and Year 2000, statutory or mandatory changes such as rate changes, and minor augmentation such as adding features that require less than a week of effort.

Table 2: U.S. Software Populations in Development and Maintenance

Year	Development Personnel	Maintenance Personnel	Total Personnel	Maintenance Percent
1950	1,000	100	1,100	9.09%
1955	2,500	250	2,750	9.09%
1960	20,000	2,000	22,000	9.09%
1965	50,000	10,000	60,000	16.67%
1970	125,000	25,000	150,000	16.67%
1975	350,000	75,000	425,000	17.65%
1980	600,000	300,000	900,000	33.33%
1985	750,000	500,000	1,250,000	40.00%
1990	900,000	800,000	1,700,000	47.06%
1995	1,000,000	1,100,000	2,100,000	52.38%
2000	750,000	2,000,000	2,750,000	72.73%
2005	775,000	2,500,000	3,275,000	76.34%
2010	800,000	3,000,000	3,800,000	78.95%
2015	1,000,000	3,500,000	4,500,000	77.78%
2020	1,100,000	3,750,000	4,850,000	77.32%
2025	1,250,000	4,250,000	5,500,000	77.27%

Notice that under the double impact of the Euro and the Year 2000 so many development projects were delayed or cancelled so that the population of software developers in the United States actually shrank below the peak year of 1995. The burst of mass update maintenance work is one of the main reasons why there is such a large shortage of software personnel.

As can be seen from table 2, the work of fixing errors and dealing with mass updates to aging legacy applications has become the dominant form of software engineering. This

tendency will continue indefinitely so long as maintenance work remains labor-intensive.

Before proceeding, let us consider 21 discrete topics that are often coupled together under the generic term "maintenance" in day to day discussions, but which are actually quite different in many important respects:

Table 3: Major Kinds of Work Performed Under the Generic Term "Maintenance"

1. Major Enhancements (new features of > 20 function points)
2. Minor Enhancements (new features of < 5 function points)
3. Maintenance (repairing defects for good will)
4. Warranty repairs (repairing defects under formal contract)
5. Customer support (responding to client phone calls or problem reports)
6. Error-prone module removal (eliminating very troublesome code segments)
7. Mandatory changes (required or statutory changes)
8. Complexity analysis (quantifying control flow using complexity metrics)
9. Code restructuring (reducing cyclomatic and essential complexity)
10. Optimization (increasing performance or throughput)
11. Migration (moving software from one platform to another)
12. Conversion (Changing the interface or file structure)
13. Reverse engineering (extracting latent design information from code)
14. Reengineering (transforming legacy application to client-server form)
15. Dead code removal (removing segments no longer utilized)
16. Dormant application elimination (archiving unused software)
17. Nationalization (modifying software for international use)
18. Year 2000 Repairs (date format expansion or masking)
19. Euro-currency conversion (adding the new unified currency to financial applications)
20. Retirement (withdrawing an application from active service)
21. Field service (sending maintenance members to client locations)

Although the 21 maintenance topics are different in many respects, they all have one common feature that makes a group discussion possible: They all involve modifying an existing application rather than starting from scratch with a new application.

Although the 21 forms of modifying existing applications have different reasons for being carried out, it often happens that several of them take place concurrently. For example, enhancements and defect repairs are very common in the same release of an evolving application. There are also common sequences or patterns to these modification activities. For example, reverse engineering often precedes reengineering and the two occur so often together as to almost comprise a linked set. For releases of large applications and major systems, the author has observed from six to 10 forms of maintenance all leading up to the same release!

Nominal Default Values for Maintenance and Enhancement Activities

The nominal default values for exploring these 21 kinds of maintenance are shown in table 4. However, each of the 21 has a very wide range of variability and reacts to a number of different technical factors, and also to the experience levels of the maintenance personnel. Let us consider some generic default estimating values for these various maintenance tasks using two useful metrics: "assignment scopes" and "production rates."

The term "assignment scope" refers to the amount of software one programmer can keep operational in the normal course of a year, assuming routine defect repairs and minor updates. Assignment scopes are usually expressed in terms of function points and the observed range is from less than 300 function points to more than 5,000 function points.

The term "production rate" refers to the number of units that can be handled in a standard time period such as a work month, work week, day, or hour. Production rates are usually expressed in terms of either "function points per staff month" or the similar and reciprocal metric, "work hours per function point."

We will also include "Lines of code per staff month" with the caveat that the results are merely based on an expansion of 100 statements per function point, which is only a generic value and should not be used for serious estimating purposes.

Table 4: Default Values for Maintenance Assignment Scopes and Production Rates

	Assignment Scopes in Function Points	Production Rates (Funct. Pts. per Month)	Production Rates (Work Hours per Funct. Pt.)	Production Rates (LOC per Staff Month)
Customer support	5,000	3,000	0.04	300,000
Code restructuring	5,000	1,000	0.13	100,000
Complexity analysis	5,000	500	0.26	50,000
Reverse engineering	2,500	125	1.06	12,500
Retirement	5,000	100	1.32	10,000
Field service	10,000	100	1.32	10,000
Dead code removal	750	35	3.77	3,500
Enhancements (minor)	75	25	5.28	2,500
Reengineering	500	25	5.28	2,500
Maintenance (defect repairs)	750	25	5.28	2,500
Warranty repairs	750	20	6.60	2,000
Migration to new platform	300	18	7.33	1,800
Enhancements (major)	125	15	8.80	1,500
Nationalization	250	15	8.80	1,500
Conversion to new interface	300	15	8.80	1,500
Mandatory changes	750	15	8.80	1,500
Performance optimization	750	15	8.80	1,500
Year 2000 repairs	2,000	15	8.80	1,500

Euro-currency conversion	1,500	15	8.80	1,500
Error-prone module removal	300	12	11.00	1,200
Average	2,080	255	5.51	25,450

Each of these forms of modification or support activity have wide variations, but these nominal default values at least show the ranges of possible outcomes for all of the major activities associated with support of existing applications.

Table 5 shows some of the factors and ranges that are associated with assignment scopes, or the amount of software that one programmer can keep running in the course of a typical year.

In table 5 the term "experienced staff" means that the maintenance team has worked on the applications being modified for at least six months and are quite familiar with the available tools and methods.

The term "good structure" means that the application adheres to the basic tenets of structured programming; has clear and adequate comments; and has cyclomatic complexity levels that are below a value of 10.

The term "full maintenance tools" implies the availability of most of these common forms of maintenance tools: 1) Defect tracking and routing tools; 2) Change control tools; 3) Complexity analysis tools; 4) Code restructuring tools; 5) Reverse engineering tools; 6) Reengineering tools; 7) Maintenance "workbench" tools; 8) Test coverage tools.

The term "high level language" implies a fairly modern programming language that requires less than 50 statements to encode 1 function point. Examples of such languages include most object-oriented languages such as Smalltalk, Eiffel, and Objective C.

By contrast "low level languages" implies language requiring more than 100 statements to encode 1 function point. Obviously assembly language would be in this class since it usually takes more than 200 to 300 assembly statements per function point. Other languages that top 100 statements per function point include many mainstream languages such as C, Fortran, and COBOL.

In between the high-level and low-level ranges are a variety of mid-level languages that require roughly 70 statements per function point, such as Ada83, PL/I, and Pascal.

The variations in maintenance assignment scopes are significant in understanding why so many people are currently engaged in maintenance of aging legacy applications. If a company owns a portfolio of 100,000 function points maintained by generalists many more people will be required than if maintenance specialists are used. If the portfolio consists of poorly structured code written in low-level languages then the assignment scope might be less than 500 function points or a staff of 200 maintenance personnel.

If the company has used complexity analysis tools, code restructuring tools, and has a staff of highly trained maintenance specialists then the maintenance assignment scope might top 3,000 function points. This implies that only 33 maintenance experts are needed, as opposed to 200 generalists. Table 5 illustrates how maintenance assignment scopes vary in response to four different factors, when each factor switches from "worst case" to "best case." Table 5 assumes Version 4.1 of the International Function Point Users Group (IFPUG) counting practices manual.

Table 5: Variations in Maintenance Assignment Scopes Based on Four Key Factors
(Data expressed in terms of function points per maintenance team member)

	Worst Case	Average Case	Best Case
Inexperienced staff Poor structure Low-level language No maintenance tools	100	200	350
Inexperienced staff Poor structure **High-level language** No maintenance tools	150	300	500
Inexperienced staff Poor structure Low-level language **Full maintenance tools**	225	400	600
Inexperienced staff **Good structure** Low-level language No maintenance tools	300	500	750
Experienced Staff Poor structure Low-level language No maintenance tools	350	575	900
Inexperienced staff **Good structure** **High-level language** No maintenance tools	450	650	1,100
Inexperienced staff **Good structure** Low-level language **Full maintenance tools**	575	800	1,400

Experienced staff **Good structure** Low-level language No maintenance tools	700	1,100	1,600
Inexperienced staff Poor structure **High-level language** **Full maintenance tools**	900	1,400	2,100
Experienced staff Poor structure Low-level language **Full maintenance tools**	1,050	1,700	2,400
Experienced staff Poor structure **High-level language** No maintenance tools	1,150	1,850	2,800
Experienced staff **Good structure** **High-level language** No maintenance tools	1,600	2,100	3,200
Inexperienced staff **Good structure** **High-level language** **Full maintenance tools**	1,800	2,400	3,750
Experienced staff Poor structure **High-level language** **Full maintenance tools**	2,100	2,800	4,500
Experienced staff **Good structure** Low-level language **Full maintenance tools**	2,300	3,000	5,000
Experienced staff **Good structure** **High-level language** **Full maintenance tools**	2,600	3,500	5,500
Average	1,022	1,455	2,278

None of the values in table 5 are sufficiently rigorous by themselves for formal cost estimates, but are sufficient to illustrate some of the typical trends in various kinds of maintenance work. Obviously adjustments for team experience, complexity of the application, programming languages, and many other local factors are needed as well.

Metrics Problems With Small Maintenance Projects

There are several difficulties in exploring software maintenance costs with accuracy. One of these difficulties is the fact that maintenance tasks are often assigned to development personnel who interleave both development and maintenance as the need arises. This practice makes it difficult to distinguish maintenance costs from development costs because the programmers are often rather careless in recording how time is spent.

Another and very signficant problem is that fact that a great deal of software maintenance consists of making very small changes to software applications. Quite a few bug repairs may involve fixing only a single line of code. Adding minor new features such as perhaps a new line-item on a screen may require less than 50 source code statements.

These small changes are below the effective lower limit for counting function point metrics. The function point metric includes weighting factors for complexity, and even if the complexity adjustments are set to the lowest possible point on the scale, it is still difficult to count function points below a level of perhaps 15 function points.

Quite a few maintenance tasks involve changes that are either a fraction of a function point, or may at most be less than 10 function points or about 1000 COBOL source code statements. Although normal counting of function points is not feasible for small updates, it is possible to use the "backfiring" method or converting counts of logical source code statements in to equivalent function points. For example, suppose an update requires adding 100 COBOL statements to an existing application. Since it usually takes about 105 COBOL statements in the procedure and data divisions to encode 1 function point, it can be stated that this small maintenance project is "about 1 function point in size."

If the project takes one work day consisting of six hours, then at least the results can be expressed using common metrics. In this case, the results would be roughly "6 staff hours per function point." If the reciprocal metric "function points per staff month" is used, and there are 20 working days in the month, then the results would be "20 function points per staff month."

Best and Worst Practices in Software Maintenance

Because maintenance of aging legacy software is very labor intensive it is quite important to explore the best and most cost effective methods available for dealing with the millions of applications that currently exist. The sets of best and worst practices are not symmetrical. For example the practice that has the most positive impact on maintenance

productivity is the use of trained maintenance experts. However the factor that has the greatest negative impact is the presence of "error –prone modules" in the application that is being maintained.

Table 6 illustrates a number of factors which have been found to exert a beneficial positive impact on the work of updating aging applications and shows the percentage of improvement compared to average results:

Table 6: Impact of Key Adjustment Factors on Maintenance (Sorted in order of maximum positive impact)

Maintenance Factors	Plus Range
Maintenance specialists	35%
High staff experience	34%
Table-driven variables and data	33%
Low complexity of base code	32%
Y2K and special search engines	30%
Code restructuring tools	29%
Reengineering tools	27%
High level programming languages	25%
Reverse engineering tools	23%
Complexity analysis tools	20%
Defect tracking tools	20%
Y2K "mass update" specialists	20%
Automated change control tools	18%
Unpaid overtime	18%
Quality measurements	16%
Formal base code inspections	15%
Regression test libraries	15%
Excellent response time	12%
Annual training of > 10 days	12%
High management experience	12%
HELP desk automation	12%
No error prone modules	10%
On-line defect reporting	10%
Productivity measurements	8%
Excellent ease of use	7%
User satisfaction measurements	5%
High team morale	5%
Sum	503%

At the top of the list of maintenance "best practices" is the utilization of full-time, trained maintenance specialists rather than turning over maintenance tasks to untrained generalists. The positive impact from utilizing maintenance specialists is one of the reasons why maintenance outsourcing has been growing so rapidly. The maintenance productivity rates of some of the better maintenance outsource companies is roughly twice that of their clients prior to the completion of the outsource agreement. Thus even

if the outsource vendor costs are somewhat higher, there can still be useful economic gains.

Let us now consider some of the factors which exert a negative impact on the work of updating or modifying existing software applications. Note that the top-ranked factor which reduces maintenance productivity, the presence of error-prone modules, is very asymmetrical. The absence of error-prone modules does not speed up maintenance work, but their presence definitely slows down maintenance work.

Error-prone modules were discovered by IBM in the 1960's when IBM's quality measurements began to track errors or bugs down to the levels of specific modules. For example it was discovered that IBM's IMS data base product contained 425 modules, but more than 300 of these were zero-defect modules that never received any bug reports. About 60% of all reported errors were found in only 31 modules, and these were very buggy indeed.

When this form of analysis was applied to other products and used by other companies, it was found to be a very common phenomenon. In general more than 80% of the bugs in software applications are found in less than 20% of the modules. Once these modules are identified then they can be inspected, analyzed, and restructured to reduce their error content down to safe levels.

Table 7 summarizes the major factors that degrade software maintenance performance. Not only are error-prone modules troublesome, but many other factors can degrade performance too. For example, very complex "spaghetti code" is quite difficult to maintain safely. It is also troublesome to have maintenance tasks assigned to generalists rather than to trained maintenance specialists.

A very common situation which often degrades performance is lack of suitable maintenance tools, such as defect tracking software, change management software, test library software, and so forth. In general it is very easy to botch up maintenance and make it such a labor-intensive activity that few resources are left over for development work. The simultaneous arrival of the year 2000 and Euro problems have basically saturated the available maintenance teams, and are also drawing developers into the work of making mass updates. This situation can be expected to last for many years, and may introduce permanent changes into software economic structures.

Table 7: Impact of Key Adjustment Factors on Maintenance (Sorted in order of maximum negative impact)

Maintenance Factors	Minus Range
Error prone modules	-50%
Embedded variables and data	-45%
Staff inexperience	-40%
High complexity of base code	-30%
No Y2K of special search engines	-28%
Manual change control methods	-27%
Low level programming languages	-25%
No defect tracking tools	-24%
No Y2K "mass update" specialists	-22%
Poor ease of use	-18%
No quality measurements	-18%
No maintenance specialists	-18%
Poor response time	-16%
Management inexperience	-15%
No base code inspections	-15%
No regression test libraries	-15%
No HELP desk automation	-15%
No on-line defect reporting	-12%
No annual training	-10%
No code restructuring tools	-10%
No reengineering tools	-10%
No reverse engineering tools	-10%
No complexity analysis tools	-10%
No productivity measurements	-7%
Poor team morale	-6%
No user satisfaction measurements	-4%
No unpaid overtime	0%
Sum	-500%

Given the enormous amount of effort that is now being applied to software maintenance, and which will be applied in the future, it is obvious that every corporation should attempt to adopt maintenance "best practices" and avoid maintenance "worst practices" as rapidly as possible.

Software Entropy and Total Cost of Ownership

The word "entropy" means the tendency of systems to detstabilize and become more chaotic over time. Entropy is a term from physics and is not a software-related word. However entropy is true of all complex systems, including software.: All known compound objects decay and become more complex with the passage of time unless effort is exerted to keep them repaired and updated. Software is no exception. The accumulation of small updates over time tends to gradually degrade the initial structure of applications and makes changes grow more difficult over time.

For software applications entropy has long been a fact of life. If applications are developed with marginal initial quality control they will probably be poorly structured and contain error-prone modules. This means that every year, the accumulation of defect repairs and maintenance updates will degrade the original structure and make each change slightly more difficult. Over time, the application will destabilize and "bad fixes" will increase in number and severity. Unless the application is restructured or fully refurbished, eventually it will become so complex that maintenance can only be performed by a few experts who are more or less locked into the application.

By contrast, leading applications that are well structured initially can delay the onset of entropy. Indeed, well-structured applications can achieve declining maintenance costs over time. This is because updates do not degrade the original structure, as happens in the case of "spaghetti bowl" applications where the structure is almost unintelligible when maintenance begins.

The total cost of ownership of a software application is the sum of four major expense elements: 1) the initial cost of building an application; 2) the cost of enhancing the application with new features over its lifetime; 3) the cost of repairing defects and bugs over the application's lifetime; 4 The cost of customer support for fielding and responding to queries and customer-reported defects.

Table 8 illustrates the total cost of ownership of three similar software applications under three alternate scenarios. Assume the applications are nominally 1000 function points in size. (To simplify the table, only a 5-year ownership period is illustrated.)

The "lagging" scenario in the left column of table 8 assumes inadequate quality control, poor code structure, up to a dozen severe error-prone modules, and significant "bad fix" injection rates of around 20%. Under the lagging scenario maintenance costs will become more expensive every year due to entropy and the fact that the application never stabilizes.

The "average" scenario assumes marginal quality control, reasonable initial code structure, one or two error-prone modules, and an average bad-fix injection rate of around 7%. Here too entropy will occur. But the rate at which the application's structure degrades is fairly slow. Thus maintenance costs increase over a five-year period, but not at a very significant annual rate.

The "leading" scenario assumes excellent quality control, very good code structure at the initial release, zero error-prone modules, and a very low bad-fix injection rate of 1% or less. Under the leading scenario, maintenance costs can actually decline over the five-year ownership period. Incidentally, such well-structured applications of this type are most likely to be found for systems software and defense applications produced by companies at or higher the Level 3 on the Software Engineering Institute (SEI) capability maturity model (CMM) scale.

Table 8: Five-Year Cost of Software Application Ownership
(Costs are in Dollars per Function Point)

	Lagging Projects	Average Projects	Leading Projects
DEVELOPMENT	$1,200.00	$1,000.00	$800.00
Year 1	$192.00	$150.00	$120.00
Year 2	$204.00	$160.00	$112.00
Year 3	$216.00	$170.00	$104.00
Year 4	$240.00	$180.00	$96.00
Year 5	$264.00	$200.00	$80.00
MAINTENANCE	$1,116.00	$860.00	$512.00
TOTAL COST	$2,316.00	$1,860.00	$1,312.00
Difference	$456.00	$0.00	-$548.00

Under the lagging scenario, the five-year maintenance costs for the application (which include defect repairs, support, and enhancements) are greater than the original development costs. Indeed, the economic value of lagging applications is questionable after about three to five years. The degradation of initial structure and the increasing difficulty of making updates without "bad fixes" tends toward negative returns on investment (ROI) within a few years.

For applications in COBOL there are code restructuring tools and maintenance workbenches available that can extend the useful economic lives of aging legacy applications. But for many languages such as assembly language, Algol, Bliss, CHILL, CORAL, and PL/I there are few maintenance tools and no commercial restructuring tools. Thus for poorly structured applications in many languages, the ROI may be marginal or negative within less than a 10 year period. Of course if the applications are vital or mission critical (such as air traffic control or the IRS income tax applications) there may be no choice but to keep the applications operational regardless of cost or difficulty.

Under the average scenario, the five-year maintenance costs for the application are slightly below the original development costs. Most average applications have a mildly positive ROI for up to 10 years after initial deployment.

Under the leading scenario with well-structured initial applications, the five-year maintenance costs are only about half as expensive as the original development costs. Yet the same volume of enhancements is assumed in all three cases. For leading applications, the ROI can stay positive for 10 to 20 years after initial deployment. This is due to the low entropy and the reduced bad-fix injection rate of the leading scenario. In other words, if you build applications properly at the start, you can get many years of

useful service. If you build them poorly at the start, you can expect high initial maintenance costs that will grow higher as time passes. You can also expect a rapid decline in return on investment (ROI).

The same kind of phenomena can be observed outside of software. If you buy an automobile that has a high frequency of repair as shown in Consumer Reports and you skimp on lubrication and routine maintenance, you will fairly soon face some major repair problems – probably before 50,000 miles.

By contrast, if you buy an automobile with a low frequency of repair as shown in Consumer Reports and you are scrupulous in maintenance, you should be able to drive the car more than 100,000 miles without major repair problems.

Summary and Conclusions

In every industry maintenance tends to require more personnel than those building new products. For the software industry the number of personnel required to perform maintenance is unusually large and may soon top 75% of all technical software workers. The main reasons for the high maintenance efforts in the software industry are the intrinsic difficulties of working with aging software, and the growing impact of "mass updates" that began with the roll-out of the Euro and the arrival of the year 2000 problem. However similar mass-updates will occur in the future as we run out of telephone numbers and social security numbers.

Given the enormous efforts and costs devoted to software maintenance, every company should evaluate and consider best practices for maintenance, and should avoid worst practices if at all possible.

References

Arnold, Robert S.; Software Reengineering; IEEE Computer Society Press, Los Alamitos, CA; 1993; ISBN 0-8186-3272-0; 600 pages.

Arthur, Lowell Jay; Software Evolution - The Software Maintenance Challenge; John Wiley & Sons, New York; 1988; ISBN 0-471-62871-9; 254 pages.

Boehm, Barry Dr.; Software Engineering Economics; Prentice Hall, Englewood Cliffs, NJ; 1981; 900 pages.

Brown, Norm (Editor); The Program Manager's Guide to Software Acquisition Best Practices; Version 1.0; July 1995; U.S. Department of Defense, Washington, DC; 142 pages.

Department of the Air Force; Guidelines for Successful Acquisition and Management of Software Intensive Systems; Volumes 1 and 2; Software Technology Support Center, Hill Air Force Base, UT; 1994.

Gallagher, R.S.; Effective Customer Support; International Thomson Computer Press, Boston, MA; 1997; ISBN 1-85032-209-0; 480 pages.

Grady, Robert B.; Practical Software Metrics for Project Management and Process Improvement; Prentice Hall, Englewood Cliffs, NJ; ISBN 0-13-720384-5; 1992; 270 pages.

Grady, Robert B. & Caswell, Deborah L.; Software Metrics: Establishing a Company-Wide Program; Prentice Hall, Englewood Cliffs, NJ; ISBN 0-13-821844-7; 1987; 288 pages.

Jones, Capers; Applied Software Measurement; McGraw Hill, 2nd edition 1996; ISBN 0-07-032826-9; 618 pages.

Jones, Capers; Critical Problems in Software Measurement; Information Systems Management Group, 1993; ISBN 1-56909-000-9; 195 pages.

Jones, Capers; Software Productivity and Quality Today -- The Worldwide Perspective; Information Systems Management Group, 1993; ISBN -156909-001-7; 200 pages.

Jones, Capers; Assessment and Control of Software Risks; Prentice Hall, 1994; ISBN 0-13-741406-4; 711 pages.

Jones, Capers; New Directions in Software Management; Information Systems Management Group; ISBN 1-56909-009-2; 150 pages.

Jones, Capers; Patterns of Software System Failure and Success; International Thomson Computer Press, Boston, MA; December 1995; 250 pages; ISBN 1-850-32804-8; 292 pages.

Jones, Capers; The Year 2000 Software Problem - Quantifying the Costs and Assessing the Consequences; Addison Wesley, Reading, MA; 1998; ISBN 0-201-30964-5; 303 pages.

Jones, Capers; Software Quality – Analysis and Guidelines for Success; International Thomson Computer Press, Boston, MA; ISBN 1-85032-876-6; 1997; 492 pages.

Jones, Capers; Software Assessments, Benchmarks, and Best Practices; Addison Wesley Longman, Boston, MA; ISBN 0-201-48542-7; 2000; 657 pages.

Jones, Capers: "Sizing Up Software;" Scientific American Magazine, Volume 279, No. 6, December 1998; pages 104-111.

Kan, Stephen H.; Metrics and Models in Software Quality Engineering; Addison Wesley, Reading, MA; ISBN 0-201-63339-6; 1995; 344 pages.

Howard, Alan (Ed.); Software Maintenance Tools; Applied Computer Research (ACR; Phoenix, AZ; 1997; 30 pages.

Marciniak, John J. (Editor); Encyclopedia of Software Engineering; John Wiley & Sons, New York; 1994; ISBN 0-471-54002; in two volumes.

McCabe, Thomas J.; "A Complexity Measure"; IEEE Transactions on Software Engineering; December 1976; pp. 308-320.

Mertes, Karen R.; Calibration of the CHECKPOINT Model to the Space and Missile Systems Center (SMC) Software Database (SWDB); Thesis AFIT/GCA/LAS/96S-11, Air Force Institute of Technology (AFIT), Wright Patterson AFB, Ohio; September 1996; 119 pages.

Muller, Monika & Abram, Alain (editors); Metrics in Software Evolution; R. Oldenbourg Vertag GmbH, Munich; ISBN 3-486-23589-3; 1995.

Multiple authors; Rethinking the Software Process; (CD-ROM); Miller Freeman, Lawrence, KS; 1996. (This is a new CD ROM book collection jointly produced by the book publisher, Prentice Hall, and the journal publisher, Miller Freeman. This CD ROM disk contains the full text and illustrations of five Prentice Hall books: Assessment and Control of Software Risks by Capers Jones; Controlling Software Projects by Tom DeMarco; Function Point Analysis by Brian Dreger; Measures for Excellence by Larry Putnam and Ware Myers; and Object-Oriented Software Metrics by Mark Lorenz and Jeff Kidd.)

Parikh, Girish; Handbook of Software Maintenance; John Wiley & Sons, New York; 1986; ISBN 0-471-82813-0; 421 pages.

Pigoski, Thomas M.; Practical Software Maintenance - Best Practices for Managing Your Software Investment; IEEE Computer Society Press, Los Alamitos, CA; 1997; ISBN 0-471-17001-1; 400 pages.

Putnam, Lawrence H.; Measures for Excellence -- Reliable Software On Time, Within Budget; Yourdon Press - Prentice Hall, Englewood Cliffs, NJ; ISBN 0-13-567694-0; 1992; 336 pages.

Putnam, Lawrence H and Myers, Ware.; Industrial Strength Software - Effective Management Using Measurement; IEEE Press, Los Alamitos, CA; ISBN 0-8186-7532-2; 1997; 320 pages.

Rubin, Howard; Software Benchmark Studies For 1997; Howard Rubin Associates, Pound Ridge, NY; 1997.

Sharon, David; Managing Systems in Transition - A Pragmatic View of Reengineering Methods; International Thomson Computer Press, Boston, MA; 1996; ISBN 1-85032-194-9; 300 pages.

Shepperd, M.: "A Critique of Cyclomatic Complexity as a Software Metric"; Software Engineering Journal, Vol. 3, 1988; pp. 30-36.

Stukes, Sherry, Deshoretz, Jason, Apgar, Henry and Macias, Ilona; Air Force Cost Analysis Agency Software Estimating Model Analysis ; TR-9545/008-2; Contract F04701-95-D-0003, Task 008; Management Consulting & Research, Inc.; Thousand Oaks, CA 91362; September 30 1996.

Symons, Charles R.; Software Sizing and Estimating – Mk II FPA (Function Point Analysis); John Wiley & Sons, Chichester; ISBN 0 471-92985-9; 1991; 200 pages.

Takang, Armstrong and Grubh, Penny; Software Maintenance Concepts and Practice; International Thomson Computer Press, Boston, MA; 1997; ISBN 1-85032-192-2; 256 pages.

Zvegintzov, Nicholas; Software Management Technology Reference Guide; Dorset House Press, New York, NY; ISBN 1-884521-0; 1994; 240 pages.