Geriatric Issues of Aging Software

Capers Jones
Software Productivity Research, LLC.

Software has been a mainstay of business and government operations for more than 50 years. As a result, all large enterprises utilize aging software in significant amounts. Some companies exceed 5,000,000 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 and increase its entropy. Aging software may also contain troublesome regions with very high error densities called error-prone modules. Repairs to aging software suffer from a phenomenon called bad fix injection, or new defects are accidentally introduced as a byproduct of fixing previous defects.

As the 21st century advances, more than 50 percent 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.

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 23 forms of modification to existing applications. The two most common meanings of the word maintenance include the following: 1) defect repairs, and 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.

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, using 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. An organization that is proactive in using geriatric tools and services can spend less than 30 percent of its annual software budget on various forms of maintenance, while an organization that has not used any of the geriatric tools and services can top 60 percent of its annual budget on various forms of maintenance.

The kinds of maintenance tools used by lagging, average, and leading organizations are shown in Table 2. Table 2 is part of a larger study that examined many different kinds of software engineering and project management tools [1].

It is interesting that the leading companies in terms of maintenance sophistication not only use more tools than the laggards, but they use more of their features as well. Again, the function point values in Table 2 refer to the capabilities of the tools that are used in day-to-day maintenance operations. The leaders not only use more tools, but they do more with them.

Before proceeding, let us consider 23 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 [2] (See Table 3 for the list of 23 topics).

Although the 23 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.

Each of the 23 forms of modifying existing applications has a different reasons for being carried out. However, 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 between six and 10 forms of maintenance all leading up to the same release.

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

Geriatric Problems of Aging Software

Once software is put into production it continues to change in three important ways:

1. Latent defects still present at release must be found and fixed after deployment.
2. Applications continue to grow and add new features at a rate of between 5 percent and 10 percent per calendar year, due either to changes in business needs or to new laws and regulations, or both.
3. The combination of defect repairs and enhancements tends to gradually degrade the structure and increase the complexity of the application. The term for this increase in complexity over time is called *entropy*. The average rate at which software entropy increases is about 1 percent to 3 percent per calendar year.

Because software defect removal and quality control are imperfect, there will always be bugs or defects to repair in delivered software applications. The current U.S. average for defect removal efficiency is only about 85 percent of the bugs or defects introduced during development [3] and has stayed almost the same for more than 10 years. The actual values are about five bugs per function point created during development. If 85 percent of these are found before release, about 0.75 bugs per function point will be released to customers. For a typical application of 1,000 function points or 100,000 source code statements, that implies about 750 defects present at delivery. About one-third – or 250 defects – will be serious enough to stop the application from running or create erroneous outputs.

Since defect potentials tend to rise with the overall size of the application, and since defect removal efficiency levels tend to decline with the overall size of the application, the overall volume of latent defects delivered with the application rises with size. This explains why super-large applications in the range of 100,000 function points, such as Microsoft Windows and many enterprise resource planning (ERP) applications, may require years to reach a point of relative stability. These large systems are delivered with thousands of latent bugs or defects.

Not only is software deployed with a significant volume of latent defects, but a phenomenon called *bad fix injection* has been observed for more than 50 years. Roughly 7 percent of all defect repairs will contain a new defect that was not there before. For very complex and poorly structured applications, these bad-fix injections have topped 20 percent [3].

Maintenance Engineering	Lagging	Average	Leading
Reverse engineering		1,000	3,000
Reengineering		1,250	3,000
Code restructuring			1,500
Configuration control	500	1,000	2,000
Test support		500	1,500
Customer support		750	1,250
Debugging tools	750	750	1,250
Defect tracking	500	750	1,000
Complexity analysis			1,000
Mass update search engines		500	1,000
Function point subtotal	*1,750*	*6,500*	*16,500*
Number of tools	*3*	*8*	*10*

Table 2: *Numbers and Size Ranges of Maintenance Engineering Tools (Size data expressed in terms of function point metrics)*

In the 1970s, IBM did a distribution analysis of customer-reported defects against their main commercial software applications. The IBM personnel involved in the study, including the author, were surprised to find that defects were not randomly distributed through all of the modules of large applications [4].

In the case of IBM's main operating system, about 5 percent of the modules contained just over 50 percent of all reported defects. The most extreme example was a large database application, where 31 modules out of 425 contained more than 60 percent of all customer-reported bugs. These troublesome areas were known as *error-prone modules*.

Similar studies by other corporations such as AT&T and ITT found that error-prone modules were endemic in the software domain. More than 90 percent of applications larger than 5,000 function points were found to contain error-prone modules in the 1980s and early 1990s. Summaries of the error-prone module data from a number of companies was published in [3].

Fortunately, it is possible to surgically remove error-prone modules once they are identified. It is also possible to prevent them from occurring. A combination of defect measurements, formal design inspections, formal code inspections, and formal testing and test-coverage analysis have proven to be effective in preventing error-prone modules from coming into existence [5].

Today in 2007, error-prone modules are almost nonexistent in organizations

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

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 or structural analysis (charting control flow plus 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 modern forms).
15. Dead code removal (removing segments no longer utilized).
16. Dormant application elimination (archiving unused software).
17. Nationalization (modifying software for international use).
18. Mass updates such as the Euro or Year 2000 (Y2K) repairs.
19. Refactoring, or reprogramming, applications to improve clarity.
20. Retirement (withdrawing an application from active service).
21. Field service (sending maintenance members to client locations).
22. Reporting bugs or defects to software vendors.
23. Installing updates received from software vendors.

that are higher than Level 3 on the Software Engineering Institute's Capability Maturity Model® (CMM®). However, they remain common and troublesome for Level 1 organizations and for organizations that lack sophisticated quality measurements and quality control.

If the author's clients are representative of the United States as a whole, more than 50 percent of U.S. companies still do not utilize the CMM at all. Of those who do use the CMM, less than 15 percent are at Level 3 or higher. That implies that error-prone modules may exist in more than half of all large corporations and in a majority of state government software applications as well.

Once deployed, most software applications continue to grow at annual rates of between 5 percent and 10 percent of their original functionality. Some applications, such as Microsoft Windows, have increased in size by several hundred percent over a 10-year period.

The combination of continuous growth of new features coupled with continuous defect repairs tends to drive up the complexity levels of aging software applications. Structural complexity can be measured via metrics such as cyclomatic and essential complexity using a number of commercial tools. If complexity is measured on an annual basis and there is no deliberate attempt to keep complexity low, the rate of increase is between 1 percent and 3 percent per calendar year.

However – and this is an important fact – the rate at which entropy or complexity increases is directly proportional to the initial complexity of the application. For example, if an application is released with an average cyclomatic complexity level of less than 10, it will tend to stay well structured for at least five years of normal maintenance and enhancement changes.

But if an application is released with an average cyclomatic complexity level of more than 20, its structure will degrade rapidly and its complexity levels might increase by more than 2 percent per year. The rate of entropy and complexity will even accelerate after a few years.

As it happens, both bad-fix injections and error-prone modules tend to correlate strongly (although not perfectly) with high levels of complexity. A majority of error-prone modules have cyclomatic complexity levels of 10 or higher. Bad-fix injection levels for modifying high-complexity applications are often higher than 10 percent.

In the late 1990s, a special kind of geriatric issue occurred which involved making simultaneous changes to thousands of software applications. The first of these *mass update* geriatric issues was the deployment of the Euro currency, which required changes to currency conversion routines in thousands of applications. The Euro was followed almost immediately by the dreaded Y2K (Year 2000) problem [6], which also involved mass updates of thousands of applications. More recently in March of 2007, another such issue occurred when the starting date of daylight savings time was changed.

Future mass updates will occur later in the century when it may be necessary to add another digit to telephone numbers or area codes. Yet another and very serious mass update will occur if it becomes necessary to add digits to social security numbers in the second half of the 21st century. There is also the potential problem of the Unix time clock expiration in 2038.

Table 4: *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%
Test coverage tools and analysis	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%
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%
Online defect reporting	10%
Productivity measurements	8%
Excellent ease of use	7%
User satisfaction measurements	5%
High team morale	5%
Sum	*503%*

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 interweave 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 significant problem is the 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 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 [7].

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 1,000 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 into 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 one function point, it can be stated that this small maintenance project is *about one 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 six 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 labor intensive, it is quite important to explore the best and most cost effective methods available for dealing with the millions of applications that cur-

® Capability Maturity Model and CMM are registered in the U.S. Patent and Trademark Office by Carnegie Mellon University.

rently 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 4 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.

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 the untrained generalists. Trained maintenance specialists are found most often in two kinds of companies: 1) large systems software producers such as IBM, and 2) large maintenance outsource vendors. The curricula for training maintenance personnel can include more than a dozen topics and the training periods range from two weeks to a maximum of about four weeks.

Since training of maintenance specialists is the top factor, Table 5 shows a modern maintenance curriculum such as those found in large maintenance outsource companies.

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 that exert a negative impact on the work of updating or modifying existing software applications. Note that the top-ranked factor that 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.

In general, more than 80 percent of latent bugs found by users in software applications are reported against less than 20 percent 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.

Software Maintenance Courses	Days	Sequence
Error-Prone Module Removal	2.00	1
Complexity Analysis and Reduction	1.00	2
Reducing Bad Fix Injections	1.00	3
Defect Reporting and Analysis	0.50	4
Change Control	1.00	5
Configuration Control	1.00	6
Software Maintenance Workflows	1.00	7
Mass Updates to Multiple Applications	1.00	8
Maintenance of Commercial Off-The-Shelf Packages	1.00	9
Maintenance of ERP Applications	1.00	10
Regression Testing	2.00	11
Test Library Control	2.00	12
Test Case Conflicts and Errors	2.00	13
Dead Code Isolation	1.00	14
Function Points for Maintenance	0.50	15
Reverse Engineering	1.00	16
Reengineering	1.00	17
Refactoring	0.50	18
Maintenance of Reusable Code	1.00	19
Object-Oriented Maintenance	1.00	20
Maintenance of Agile and Extreme Code	1.00	21
TOTAL	*23.50*	

Table 5: *Sample Maintenance Curricula for Companies Using Maintenance Specialists*

Table 6 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 common situation that 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 easy to botch-up maintenance and make it such a labor-intensive activity that few resources are left over for development work.

The last factor in Table 6, no unpaid overtime, deserves a comment. Unpaid overtime is common among software maintenance and development personnel. In some companies it amounts to about 15 percent of the total work time. Because it is unpaid it is usually unmeasured. That means side-by-side comparisons of productivity rates or costs between groups with unpaid overtime and groups without will favor the group with unpaid overtime because so much of their work is uncompensated and, hence, invisible. This is a benchmarking trap for

Maintenance Factors	Minus Range
Error-prone modules	-50%
Embedded variables and data	-45%
Staff inexperience	-40%
High complexity of base code	-30%
Lack of test coverage analysis	-28%
Manual change control methods	-27%
Low-level programming languages	-25%
No defect tracking tools	-24%
No *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%*

Table 6: I*mpact of Key Adjustment Factors on Maintenance (sorted in order of maximum negative impact)*

the unwary. Because excessive overtime is psychologically harmful if continued over long periods, it is unfortunate that unpaid overtime tends to be ignored when benchmark studies are performed.

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 destabilize 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, it eventually 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 six 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, 5) the cost of periodic restructuring or *refactoring* of aging applications to reduce entropy and thereby reduce bad-fix injection rates, and 6) removal of error-prone modules via surgical removal and redevelopment. This last expense element will only occur for legacy applications that contain error-prone modules.

Similar phenomena can be observed outside of software. Hypothetically, 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 – usually well 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 building new products. For the software industry, the number of personnel required to perform maintenance is unusually large and may soon top 70 percent 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. Special factors such as *mass updates* that began with the roll-out of the Euro and the Y2K problem are also geriatric issues.

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

1. Jones, Capers. "Analyzing the Tools of Software Engineering." Software Productivity Research (SPR) Technical Report. Burlington, MA: 1999.
2. Jones, Capers. Estimating Software Costs. 2nd ed. McGraw Hill, 1998.
3. Jones, Capers. Software Quality – Analysis and Guidelines for Success. Boston, MA: International Thomson Computer Press, 1997.
4. Jones, Capers. "Program Quality and Programmer Productivity." IBM Technical Report. TR 02.764. San Jose, CA: IBM, 1977.
5. Jones, Capers. Software Assessments, Benchmarks, and Best Practices. Boston, MA: Addison Wesley Longman, 2000.
6. Jones, Capers. The Year 2000 Software Problem – Quantifying the Costs and Assessing the Consequences. Reading, MA: Addison Wesley, 1998.
7. Jones, Capers. Applied Software Measurement. 2nd ed. McGraw Hill, 1996.

Additional Reading

1. Arnold, Robert S. Software Reengineering. IEEE. Los Alamitos, CA: Computer Society Press, 1993.
2. Arthur, Lowell Jay. Software Evolution – The Software Maintenance Challenge. New York: John Wiley & Sons, 1988.
3. Gallagher, R.S. Effective Customer Support. Boston, MA: International Thomspon Computer Press, 1997.
4. Kan, Stephen H. Metrics and Models in Software Quality Engineering. Reading, MA: Addison Wesley, 2003.

About the Author

Capers Jones is currently the chairman of Capers Jones and Associates, LLC. He is also the founder and former chairman of SPR, where he holds the title of Chief Scientist Emeritus. He is a well-known author and international public speaker, and has authored the books "Patterns of Software Systems Failure and Success," "Applied Software Measurement," "Software Quality: Analysis and Guidelines for Success," "Software Cost Estimation," and "Software Assessments, Benchmarks, and Best Practices." Jones and his colleagues from SPR have collected historical data from more than 600 corporations and more than 30 government organizations. This historical data is a key resource for judging the effectiveness of software process improvement methods. The total volume of projects studied now exceeds 12,000.

Software Productivity Research, LLC
Phone: (877) 570-5459
Fax: (781) 273-5176
E-mail: capers.jones@spr.com, info@spr.com