Software Total Ownership Costs: Development Is Only Job One

The 10 step process includes rigorous estimation, measurement and lessons learned rather than the haphazard estimation that is all too often the case.

by Daniel D. Galorath, Galorath Inc.

Planning software development projects is never an easy undertaking. Issues such as customer and competitive requirements, time-to-market, architectural and quality considerations, staffing levels and expertise, potential risks, and many other factors must be carefully weighed and considered. Software development costs only comprise a portion – often the smaller portion – of the total cost of software ownership. However, the development process itself has a significant impact on total cost of ownership as tradeoffs are evaluated and compromises made that impact sustainability and maintainability of software over time.

During maintenance, accumulation of poorly managed changes almost always generates software instability and a significant increase in the cost of software maintenance – up to 75% of just the software total ownership costs, according to some estimates. On top of that, IT services and infrastructure can comprise another 60%[i] of the total cost to the enterprise, in addition to the software development costs -- and these costs are rarely considered by development managers.

The findings come from observing thousands of systems over several decades and looking at the difficulty organizations have with affordability in software maintenance and total costs. Development managers generally do not manage for total ownership costs, but rather to get the development completed.
This paper discusses total cost of ownership across the software life cycle, including estimation, measurement criteria, metrics, industry standards, guidelines, and best practice options. Parametric modeling is included with specific examples.

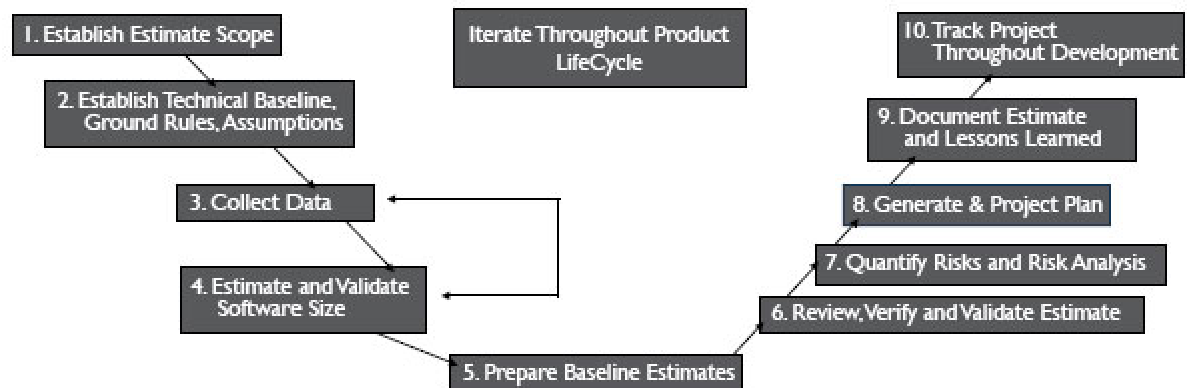


Figure 1: 10 Step Software Estimation Process

10 Step Estimation Process Improves Projects As Well As Estimates

More projects fail due to planning issues than technical or any other issues[ii]. Better project planning is key to successful development projects, and maintenance, and total ownership cost. In that regard, the 10 step software estimation process[iii] is proposed. Continuing to apply this estimation process to software development, IT infrastructure

and IT services, operations and software maintenance can make great strides in improving software total ownership costs. The 10 step process is illustrated in Figure 1. The 10 step process includes rigorous estimation, measurement and lessons learned rather than the haphazard estimation that is all too often the case. Many people ask how the estimation process can help project and total ownership costs. True estimation assumes both careful analysis (not just quick guessing) with the application of the 10 step process and the use of parametric estimation techniques that allow the statement and capturing of assumptions and plans encompassing people, process, and technology as well as constraints.

As one industry editor put it, “the journey is its own reward.” Merely considering the issues up front will make the project planning better. For example, a project parameter regarding specification level, test level, or quality assurance level can provide insight not only into effort and schedule during development, but also delivered defects and product quality delivered to the maintainers.

Major process models such as CMMI call out estimation and planning, measurement and analysis, and monitoring and control as key process areas. The 10 step process encourages the symbiotic relationship between estimation, measurement, monitoring and control. The monitoring and control aspects are really more measurement and adjustment, including re-estimating based on what the current and anticipated reality is. Estimation is key to planning. Measurement is key to controlling and improving estimation. And using estimation to ensure planning during development, then applying estimation practices during maintenance has a significant ROI when tradeoffs are made to reduce total ownership cost. Yet, many skimp on this analysis so they can get what they consider “the real work” done.

Measurement Provides Insight and Management Potential Throughout the Product Life Cycle

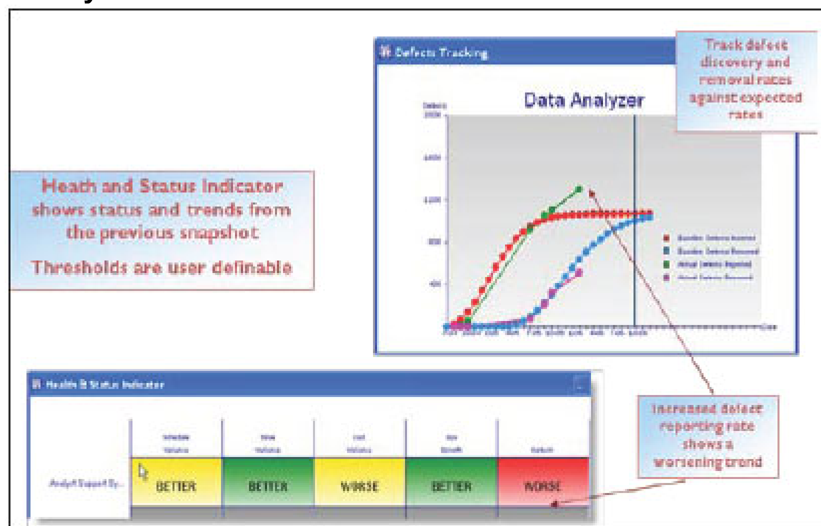


Figure 2: Example Reporting Defect Removal Effectiveness Metrics & Performance Based Earned Value

From the early 20th century when Frederick Taylor founded the principals of scientific management ("let the data and facts do the talking"), to W. Edwards Demming ("In God We Trust, All Others Bring Data"), to Eli Goldratt[iv] ("Improvements should increase profit and effectiveness"), measurement gurus have stressed the importance of measuring and using such information to estimate and manage. In studying and applying metrics over the years, and thanks in part to Eli Goldratt , it becomes apparent that there are two types of metrics, 1) the obvious status and trend metrics such as productivity, defect removal rate, cost, schedule, etc. and 2) the potentially not so obvious effectiveness metrics. Effectiveness metrics may change over time as internal process improvements remove problems, making some of them obsolete and others apparent. They are, essentially, what are we doing that we should not do and what are we not doing that we should do. According to Goldratt, there should be no more than 5 effectiveness metrics. For example, if we are not finding and removing defects at an appropriate rate, testing is a candidate for measurement and improvement. Figure 2 illustrates defect insertion / removal effectiveness. In Figure 2 we see the rate of defect insertion is higher than what was predicted and the rate of defect removal is lower than anticipated. The good news is that we are measuring and finding the defects and that we may have insight into customer satisfaction issues.

Many projects don't even find them. Nevertheless, there appears to be something we are not doing that we should be doing in our processes or reporting. Defect prone software can cost more, both during development and maintenance.

Today, many projects use earned value at a very high level, tracking the overall program performance in terms of effort and progress. This is a good thing since earned value is a powerful program management technique, combining measurement and estimation (baseline plan) to determine progress compared to the plan. However, for software projects (both development and maintenance) this is only part of the picture. Additional dimensions (Four Dimensional Earned Value[v]) includes traditional earned value effort and progress measures and also takes into account the idiosyncrasies of software projects: size growth and defect insertion & removal issues. In software, premature victory declarations[vi] such as claiming progress when testing is not sufficient or when the scope is creeping beyond the baseline plan, result in the project realizing the trouble later. So, many times we see projects where peer reviews or inspections are skipped in the name of productivity or catching up. Scope creep, increasing requirements during development, is nearly the norm. Both should be tracked in addition to effort and progress to get the full picture in a software project, where it really is, and where it is going. Again, these techniques are valuable not only during development but also during maintenance innovations (additional functionality added during maintenance.) Additionally, risk analysis is an essential component of software planning and software management that is not generally supported by high level earned value management (EVM).

Software Maintenance; The Larger Side of the Equation
There are numerous standards and definitions for software maintenance. The IEEE 1219 definition is representative: “The modification of a software product after delivery to correct faults, to improve performance or other attributes, or to adapt the product to a modified environment.” It even appears that poor quality is cheaper until the end of coding. Figure 3, which comes from an article by Capers Jones titled “Software Quality - a Survey of the State of the Art”, (published by Productivity Research LLC, Narragansett, RI; 2008), shows that poor quality is cheaper until the coding activities are complete, then higher after that in development and maintenance.

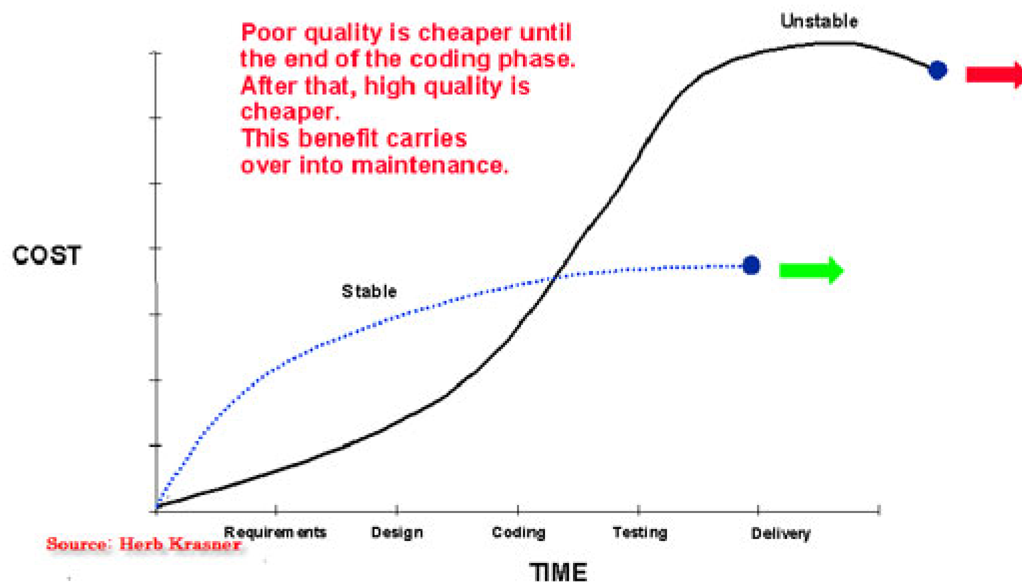


Figure 3: The Early Maintenance Benefit of Quality
Herb Krasner[vii] also points out that software evolution (that is, increasing functionality to meet new requirements) is inevitable, expensive and difficult, and that it is possible to increase the maintainability of a software system by designing it properly in the first place.

Maintenance typically accounts for 75% or more of the total software workload. The cost is driven by the development quality and highly dependent on maintenance rigor and operational “life expectancy.” The activities of maintenance (Figure 4 Maintenance Activities Summarized) generally include sustaining engineering and new function development.
• Corrective changes (e.g. fixing defects)
• Adapting to new requirements (e.g. OS upgrade, new processor)
• Perfecting or improving existing functions (e.g. improve speed, performance)
• Enhancing application with new / innovative functions

Maintenance Critical Success Factors

Maintenance is difficult for most systems. There are many reasons for this, the most common being that maintenance is not a goal of development. Items like inadequate documentation, less experienced staff, lack of quality assurance or test rigor, and the tendency to produce quick and dirty fixes (both during maintenance and development) all work together to make software maintenance[viii] exceedingly expensive and difficult.

Identifying and managing critical success factors in software maintenance yield enduring software that does not degrade due to maintenance. Independent of development sins, if a project can achieve the following critical success factors during maintenance, significant cost reductions can be achieved.

Factors include:

- Functionality: Preserve or enhance functionality
- Quality: Preserve or increase quality of system
- Complexity: Should not increase product complexity relative to the size
- Volatility: Should not lead to increase in product volatility
- Costs: Relative costs per maintenance task should not increase for similarly scoped tasks
- Deadlines: Agreed upon release deadlines should be kept and delays should not increase
- User Satisfaction: Increase or at least not decrease
- Profitability: Be profitable or at least cover its costs

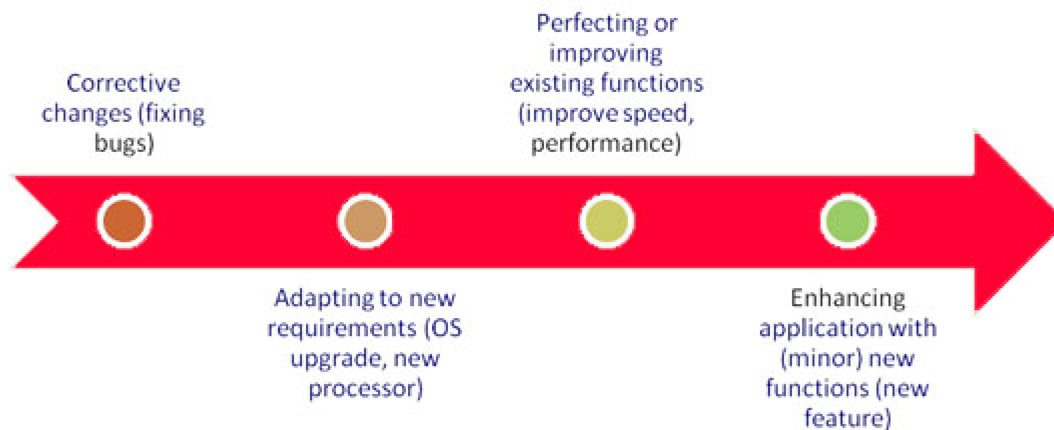
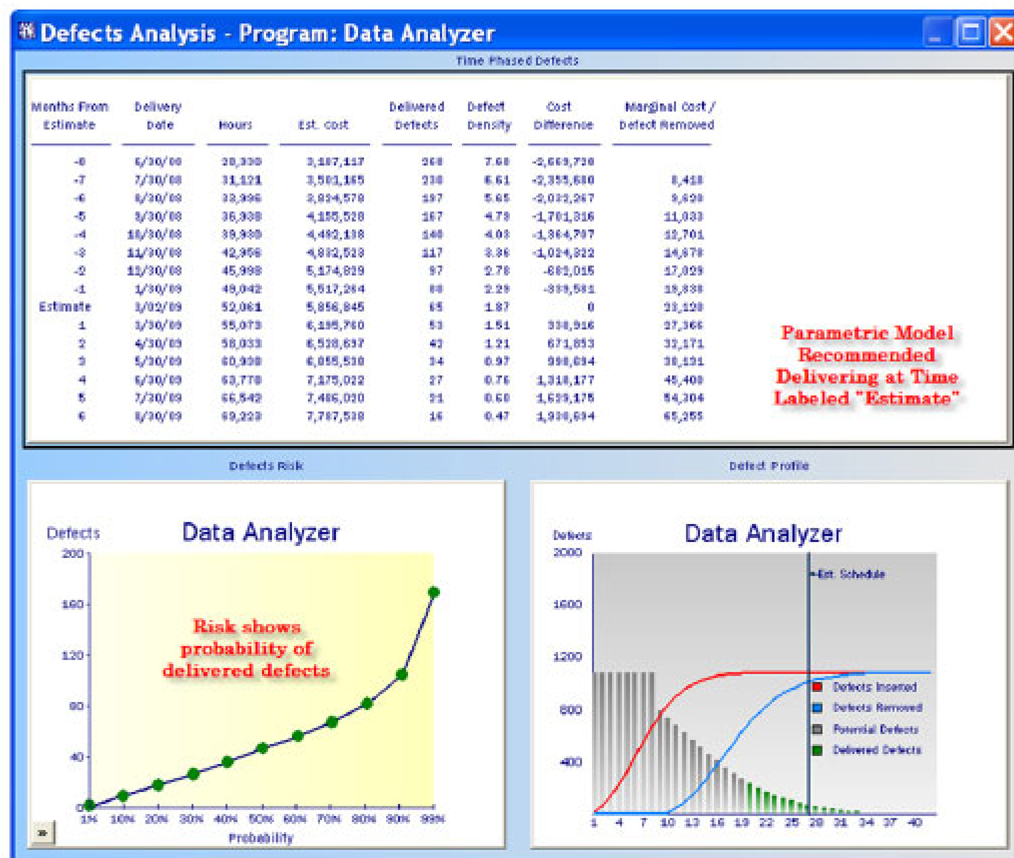


Figure 4: Maintenance Activities Summarized

Software Maintenance Metrics

Meaningful software metrics are only effective if sufficient staff is available for maintenance. Often maintenance is treated as a level of effort activity rather than dedicating adequate resources to protect the software investment. In such cases software will naturally degrade. The author has seen situations as bad as one person maintaining a million lines of code. That program just about shut down during maintenance, but the program office felt good about all the money they "saved." It is not known whether root cause analysis was performed when the system had to be scrapped and redeveloped, but it is likely this severe understaffing of maintenance would have been the identified culprit.

Understanding when a project transitions from development to maintenance can provide clues regarding maintainability
as well as project success. Fielding a software product before it has been stabilized is the cause of many project failures[ix] . For example, in Figure 5 at the top and bottom right we can see the earliest time possible to transition the product to maintenance: March 2009 with 65 delivered defects. If deployed earlier, the number of defects will likely destroy confidence and any chance for future success. Again from the chart, deploying 5 months earlier could deliver 167 defects. While 65 defects may sound terrible and will certainly keep the staff hopping during early deployment, 167 is a catastrophe. If deployed 5 months later, delivered defects drop, but costs skyrocket, [x] and the software isn't providing the business value to the organization until later. Of course there is risk and uncertainty as well, as shown by the bottom graphic.



Time Phased Defects

Months From Estimate	Delivery Date	Hours	Est. Cost	Delivered Defects	Defect Density	Cost Difference	Marginal Cost / Defect Removed
-8	6/30/08	28,330	3,187,117	268	7.68	-2,669,728	
-7	7/30/08	31,121	3,501,165	230	6.61	-2,355,680	8,418
-6	8/30/08	33,996	3,824,578	197	5.65	-2,032,267	9,628
-5	9/30/08	36,938	4,155,528	167	4.79	-1,701,316	11,033
-4	10/30/08	39,930	4,492,138	140	4.03	-1,364,707	12,701
-3	11/30/08	42,956	4,832,523	117	3.36	-1,024,322	14,678
-2	12/30/08	45,998	5,174,829	97	2.78	-682,015	17,029
-1	1/30/09	49,042	5,517,264	80	2.29	-339,581	19,838
Estimate	3/02/09	52,061	5,856,845	65	1.87	0	23,120
1	3/30/09	55,073	6,195,760	53	1.51	338,916	27,366
2	4/30/09	58,033	6,528,698	42	1.21	671,853	32,171
3	5/30/09	60,930	6,855,538	34	0.97	998,694	38,131
4	6/30/09	63,770	7,175,022	27	0.76	1,318,177	45,400
5	7/30/09	66,542	7,486,020	21	0.60	1,629,175	54,204
6	8/30/09	69,223	7,787,538	16	0.47	1,930,694	65,255

Figure 5: Software Defect Analysis

Example Maintenance Metrics

Generally, you get what you measure. Choosing the right metrics for your project is a key consideration.

The following identifies some of the possible traditional appropriate maintenance metrics:

• Defects removed per unit time

• Productivity for block changes
• Mean time to find the next k faults
• Maintenance backlog
• Increases / decrease on maintenance backlog
• Number of trouble reports opened and closed
• Mean time until problem closed
• Defects during warranty period
• Mean time to resolution
• Defects by type and severity
• Time to respond to customer reported defects
• McCabe & Halstead complexity metricsxi
• Software Maturity Index (IEEE 982 Standard Dictionary of Measures To Produce Reliable Software)
M = number of modules in current version
A = number of added modules in current version
C = number of changed modules in current version
D = number of deleted modules in current version compared to the previous version
SoftwareMaturityIndex = (M - (A + C + D)) / M
When the Software Maturity Index approaches 1.0 the product is stable. Unfortunately, in many domains,
it is likely old enough to be approaching retirement at this point.

Example Maintenance Effectiveness Metrics

These are examples of metrics that may identify what we are doing that we should not and/or what we are not doing that we should:
• Number of new defects created by fixes
• Number of defect corrections that were not correct (Defective)
• Number of defects not repaired in promised time (Delinquent)
• Defect Seepage (Customer reported defects during pre-delivery testing)

Maintenance Productivity Drivers

When looking at software maintenance costs, the sins or blessings of development come into play, plus a number of other items that help scope maintenance.

Years of Maintenance: Number of years for which software maintenance costs will be estimated (Maintenance typically begins when operational test & evaluation is completed).

Separate Sites: Number of separate operational sites where the software will be installed and users will have an input into system enhancements (Only sites that have some formal input, not necessarily all user sites). More sites = more enhancing, corrective, and perfective effort

Maintenance Growth Over Life: Anticipated size growth from the point immediately after the software is turned over to maintenance through the end of the product life cycle. This includes additions of innovations and other functionality. Figure 6 illustrates development versus maintenance growth.

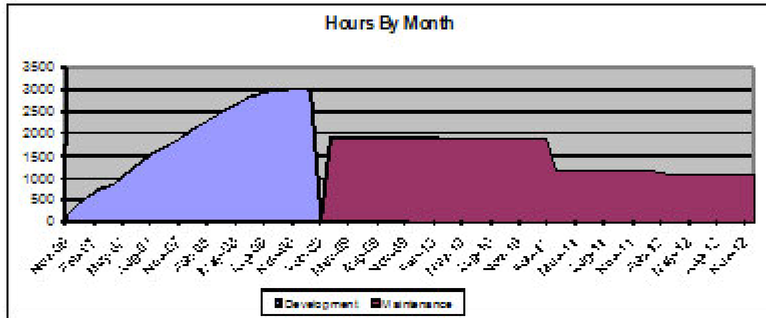


Figure 6: Development Vs Maintenace 0 Growth

Maintenance Rigor: How well the system will be cared for. As illustrated in (Figure 7), Staff Vs Maintenance Rigor, Vhi+ rigor means maintenance will be staffed to ensure protection of the software investment without trying to skimp. Rigor vlo means maintenance will be taken care of on an as needed basis, critical changes only. This is a key maintenance driver that is often overlooked during development analysis of total ownership cost.

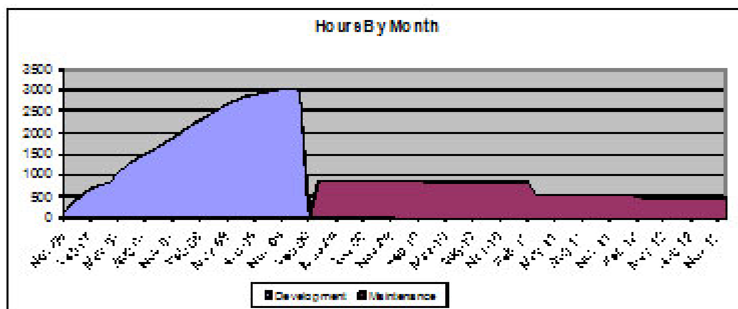


Figure 7: Development Vs. Maintenance 100% Growth

Annual Change Rate: Average percent of the software impacted by software maintenance and sustaining engineering per year. This may include changes, revalidation, reverse engineering, re-documentation, minor changes for new hardware, or recertification.

Modern Practices: The use of modern software engineering practices on the project. Practices include items such as design tools, reviews, etc.

Specification Level defines the rigor and robustness of the specifications during development and maintenance. Very low is software for personal use only. The top of the scale is tested for systems where reliability is of the utmost concern with severe penalties for failure. Lower specifications can decrease development but there is a corresponding increase in maintenance costs.

Test Level is the robustness of testing. Very low is software for personal use only. The top of the scale is tested for systems where reliability is of the utmost concern with severe penalties for failure.

Figures 8 through 12 show several of the issues related to some of the various project parameters and their impact on project effort. The "Dev" plot shows the impact during development and the "Maint" plot shows the impacts during software maintenance[xii]. These plots assume that if, for example, a high degree of specification is required during development that the same degree of specification is required during maintenance. Thus the sensitivity may be more or less than the intuitive value.

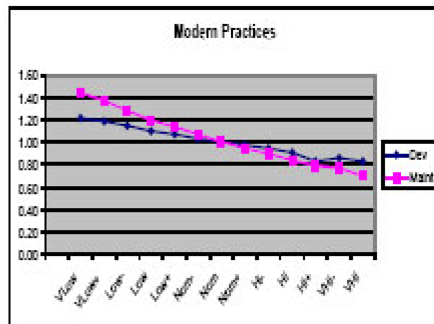


Figure 8: Development Vs. Maintenance: Use of Sound

Software Engineering Practices

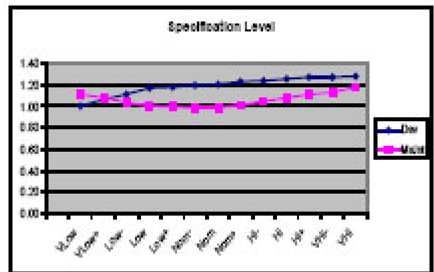


Figure 9: Staff Vs Maintenance Rigor

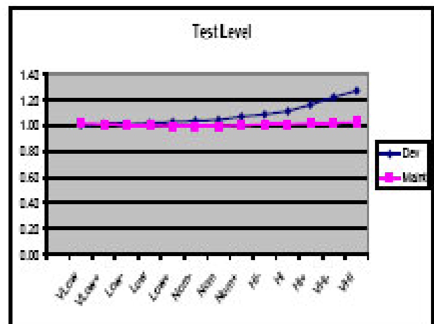


Figure 10: Development Vs Maintenance: Level of Testing

Quality Assurance Level is the robustness of quality assurance. Very low is no quality assurance. The top of the scale is tested for systems where reliability is of the utmost concern with severe penalties for failure. This is shown in Figure 11.Development Vs Maintenance Quality Assurance Level .

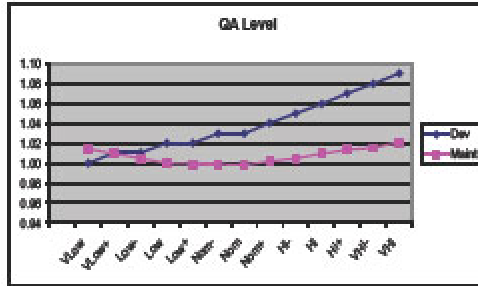


Figure 11: Development Vs Maintenance Quality Assurance Level

Personnel Capabilities and Differences From Developers describes the abilities of the personnel (as a team) that will be maintaining the software. Note industry best practices show that the maintenance team does not need to be the same as the developers[xiii], so long as strict processes and controls are in place.

Tools and Practices and Differences from Development: Many times tools and practices desired during development get put on the back burner due to delivery pressures. When maintenance is a priority these often get restored in maintenance.

Reusability: It is far more costly to build software for reuse. Maintenance must ensure the integrity of reusability. However maintenance also gets some of the benefits of the extra cost of reusability, as shown in Figure 12.

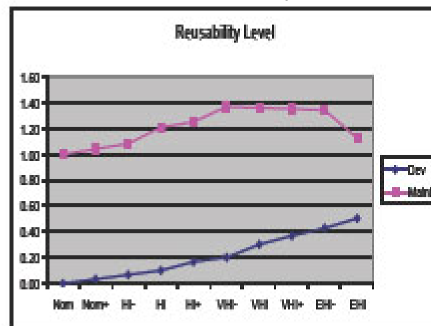


Figure 12: Development Vs. Maintenance Reusability

IT Infrastructure & Services Must Be Considered In Total Ownership Costs
When considering software total ownership costs it would be beyond remiss if the costs of IT infrastructure and IT services to deploy, migrate data, support users, and other software support issues were not considered. As discussed previously, research has shown that IT services outside the software development and maintenance (e.g. hardware cost, help desk, upgrade installation, training, etc.) can account for over 60% of the total ownership costs, (Figure 13), Figure 14 shows an IT system estimate including software, IT Infrastructure and IT Services for the project and operations & maintenance. In this example project software development (WBS 1.2) is $608,464 while the total project including infrastructure, installation, data migration, etc. is three time more costly. And looking over the project life cycle IT services dwarf the development cost with a total system cost of over $6,033,236. Just looking at software development or even software development and maintenance could lead to a very wrong view of total ownership costs. Note: In this example just the cost of IT Help desk for the 2400 users is $3.3M over the 4

years of system operation. More time during development on user interface, usability analysis and testing, and other learning aids can have a significant impact on total ownership costs by reducing help desk support; fewer releases and upgrades can reduce IT Support costs as well.

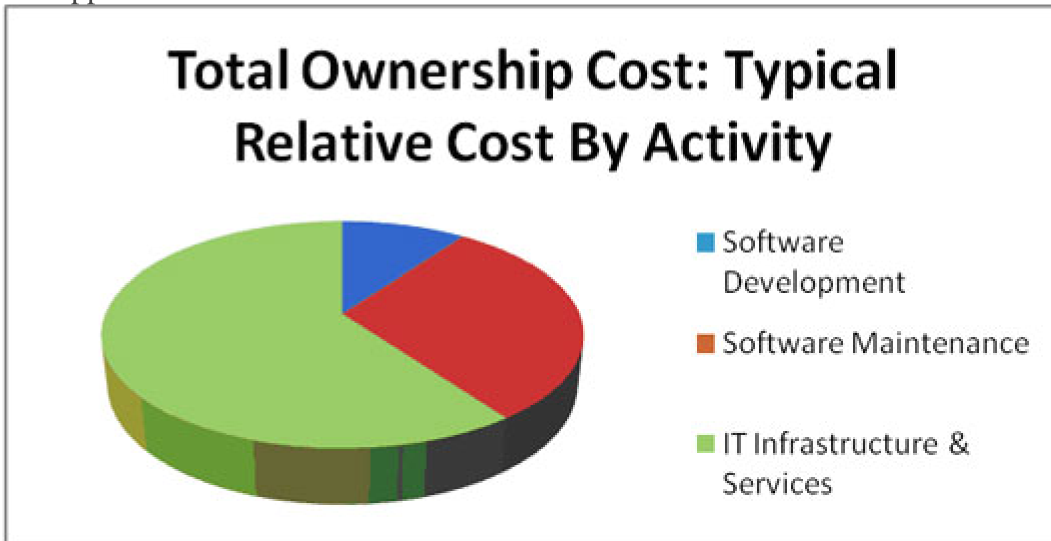


Figure 13: Relative Cost of IT Vs. Software Development Vs. Software Maintenance

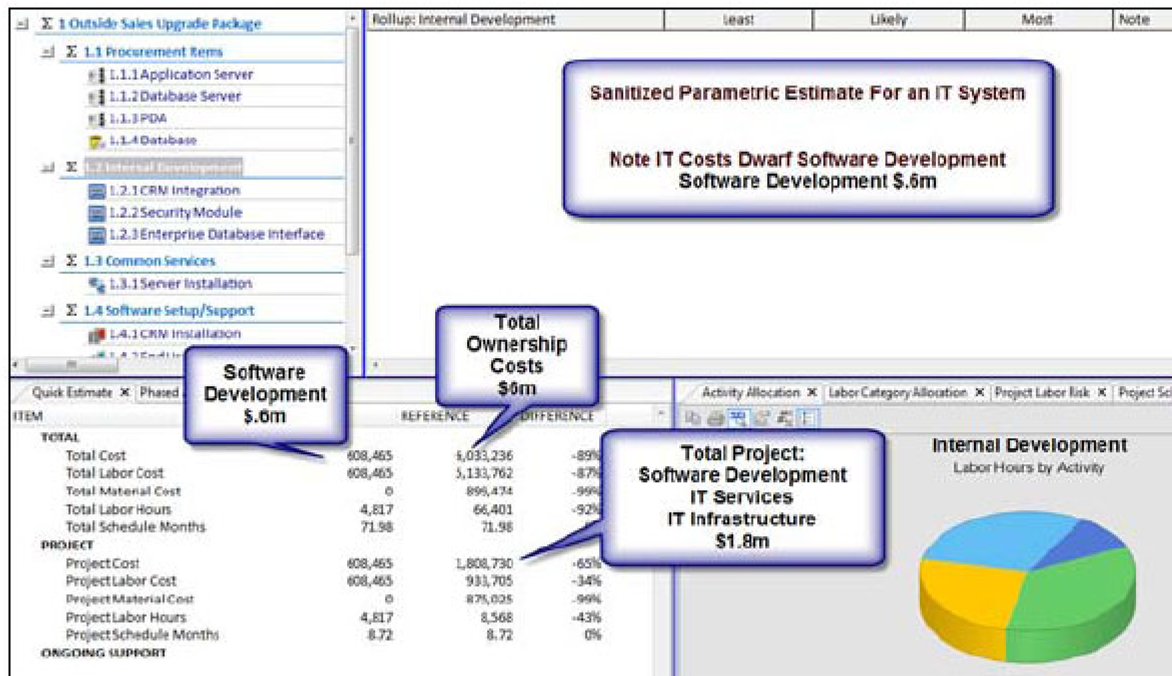


Figure 14: Example IT Support Costs For An Application

Generating an Estimate

For software, appropriate estimation can actually impact total ownership costs as decisions are made of total ownership costs of includes estimating development and maintenance. It is important to optimize these areas. While development equations for

SEER®, COCOMO, and other software development models are fairly well known, maintenance equations are not.

SEER for Software, for example uses sophisticated software total ownership model modeling including all the information from development (people, process, technology, defects, scope, etc.) and specifics about the maintenance such as the rigor and amount of the software to be maintained.

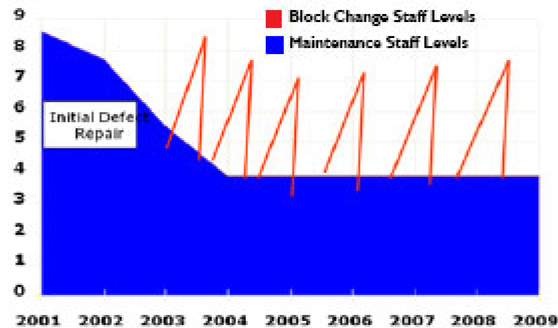


Figure 15: Maintenance Staffing Plus Block Changes

Figure 15: Maintenance Staffing Plus Block Changes, illustrates how maintenance is often higher in the early years than later, assuming the system remains relatively stable. Of course, block changes, major changes with significant functionality (innovation) often occur, causing increases in staffing. Those block changes are often estimated and managed as enhancements to existing systems, in addition to the maintenance.
The following equation[iii] (simplified to fit the scope of this paper) approximates the steady state (staff) maintenance (when the maintenance flattens):

$$\text{SteadyStateMainteanceStaff Level} = 0.393469 * \text{Complexity}^{-0.35} * ((\text{MaintSize})/\text{MaintCte})^{1.2} * \text{MaintQuality}$$

Complexity ranges from 28(vlo-) to 4 (ehi+), so complexity$^{-.35}$ increases from 0.31 to 0.61 as you increase complexity

MaintSize is how much of the system size must be maintained, usually total size, but can be effective size, adjusted from maintenance growth, change and sites.

MaintCte is the effective technology rating (People, Process, Products) using the maintenance sensitivities

MaintQuality is “Maintenance Rigor”which ranges from 0.6 to 1.2
Also a classic rough order of magnitude rule of thumb from Barry Boehm’s original COCOMO for determining annual maintenance effort can also be useful:

Annual Maintenance Effort = (Annual Change Rate) * (Original Software Development Effort)
The quantity Original Software Development Effort refers to the total effort (person-months or other unit of measure) expended throughout development, even if a multi-year project.

The multiplier Annual Change Rate is the proportion of the overall software to be modified during the year. This is relatively easy to obtain from engineering estimates. Developers often maintain change lists, or have a sense of proportional change to be required even before development is complete.

In Summary

Development decisions, processes and tools can have a significant impact on maintenance and total ownership costs. These software maintenance costs can be 75% of software total ownership costs while IT infrastructure can add another 60% on top of software. Estimation / planning processes, measurement and analysis and monitoring and control can both reduce costs themselves and can point to the areas and decisions that can reduce total ownership costs. Treating software maintenance as a level of effort activity has consequences in quality, functionality and reliability as well as costs. Applied measurement is a critical component of software and systems management.
Apply estimation techniques to determine the cost, effort, schedule, risk, stakeholder satisfaction of spending a bit more in development to reduce maintenance and total ownership costs. Apply estimation and planning, measurement and analysis, and monitoring and control to develop and maintain software with sufficient documentation and quality to optimize total costs of ownership.

About The Author

During his over three decades in the industry,**Daniel D. Galorath** of Galorath Inc., has been solving a variety of management, costing, systems, and software problems for both information technology and embedded systems. He has performed all aspects of software development and software management. One of his strengths has been reorganizing troubled software projects, assessing their progress, applying methodology and plans for completion and estimated cost to complete. He has personally managed some of these projects to successful completion. He has created and implemented software management policies, and reorganized (as well as designed and managed) development projects. He is founder and CEO of Galorath Incorporated, which has developed the SEER application for Software, Hardware, Electronics & Systems, Manufacturing, and Information Technology: cost, schedule, and risk analysis, and management decision support. He was one of the principal architects of SEER for Software (SEER-SEM) cost, schedule, risk, reliability estimation model. His teaching experience includes development and presentation of courses in Software Cost, Schedule, and Risk Analysis; Software Management; Software Engineering; and Weapons Systems Architecture. Mr. Galorath has lectured internationally. Among Mr. Galorath's published works are papers encompassing software cost modeling, testing theory, software life cycle error prediction and reduction, and software and systems requirements definition. Mr. Galorath was named winner of the 2001 International Society of Parametric Analysts (ISPA) Freiman Award, awarded to individuals who have made outstanding contributions to the theoretical or applied aspects of parametric modeling. Mr Galorath's book, "Software Sizing, Estimation, and Risk Management" was published in March 2006. His blog may be found at www.galorath.com/wp

Author Contact Information

Dan Galorath: Galorath@galorath.com

References

[i]GartnerGroup, IT Worldwide Spending, http://www.gartner.com/it/page.jsp?id=742913, July 2008.
[ii]More projects fail due to lack of planning… find references.
[iii]Galorath, D, Evans, M, Software Sizing, Estimation and Risk Management, Auerbach Publications, 2006
[iv]Goldratt,E., Theory of Constraints, North River Press, 1999.
[v]Galorath Incorporates, SEER for IT User Guide, 2008.
[vi]Evans, M, Abela, A, Beltz T, , even Characteristics of Dysfunctional Software Projects, Crosstalk, 2002
[vii]Krasner, Herb, The ROI of Software Evolution and Maintenance Best Practices: Where to Find the Benefits, 2008.
[viii]Sneed, H, Brossler, P, Critical Success Factors In Software Maintenance: A Case Study, IEE, 2003.
[ix]Standish Group, Chaos Report, 1998.
[x]Galorath, D, Reifer, D, “Using Statistics To Determine How Much Testing Is Enough”, Prepared for Naval Underwater Systems Center, Newport, RI, 1981.Reifer, Galorath…. When should you stop testing.. finish reference.
[xi]IEEE, IEEE Standard for Software Quality Assurance, , IEEE Std 730-2002, 2002.
[xii]Galorath Incorporated, SEER for Software User Guide, 2008.
Tony Salvaggio, Compaid CEO Briefing, 2008.