Software Sustainment: Pay Now or Pay Later

With today's budget constraints, the Department of Defense is likely to build fewer and fewer new systems. In light of this, the systems that we have today will need to be stretched further. This means more time and effort will be going into maintenance. And since many of the systems fielded today rely on software to deliver much of their mission critical capability; it is likely that more time and money will need to be devoted to software sustainment.

The term software sustainment refers to anything that needs to be done to keep a software program delivering its required functionality. This includes many activities; fixing bugs that are identified, adding new features, upgrading software to work in changing environments, addressing technical debt to keep the application structurally sound, retesting the application as changes are made, etc. In some environments, sustainment activities also include things like training and operational support.

One of the major drivers of software sustainment costs is the quality of the original software. High quality software has fewer bugs, needs less refactoring and is easier to understand for bug fixes, upgrades and enhancements. And there are many things that can be done during development to improve the chances of delivering quality software. These may include choosing the right programming language, instituting and following good development processes and programming practices, using CASE tools, enforcing good documentation practices, performing regression testing, automated testing and employing code reviews and static analysis to analyze the code and other artifacts of the software development process. Most of the things that are done to improve the quality of the software tend to add time and cost to the acquisition process.

The question going forward is how we can make tradeoffs during the development of software to create a balance between cost and quality that will optimize total ownership costs. The motivation for this paper comes from a research in progress studying the cost implications of software sustainment. The intent of the research is to develop comprehensive cost estimating relationships for software sustainment activities. One of the areas of focus for this study investigated how development practices influence software quality and maintenance costs.

This paper focuses on that part of the study and is structured as follows. The first section describes software maintenance and its costs and describes related research in this area. The second section discusses software quality, the characteristics of quality software and the relationships between quality and sustainment costs. The third section presents the practices which have proven to lead to quality software and reduced sustainment costs. The final section contains conclusions and recommendations.

Software Maintenance

According to [1] software maintenance is defined as the process of modifying, for update or repair, existing operational software but leaving its primary function intact. Maintenance activities are generally classified as one of the following:

- Preventive – proactively addressing known weaknesses in the code base through refactoring

- Corrective – reactively fixing defects that are discovered in the code
- Perfective – enhancing existing features and functions
- Adaptive – making changes to adapt to a changing environment, improve performance, etc.

There are quite a few factors that may influence the costs of software maintenance. Certainly logistical issues are part of the equation. What parts of the system need to be maintained, how long the system needs to be maintained and the criticality of maintenance for the entire system or subsystems (how critical is the system or subsystem to mission success)? Who is expected to do the sustainment activities – will it be done by the original development team, a new team or even an entirely different organization? Will the sustainment team have access to the same development tools, test cases, regression tests and testing environments? How much of the software system is comprised of Commercial Off the Shelf (COTS) solutions, what are the interfaces and how well are those interfaces documented?

In addition to the logistical factors cited above, there are also technical factors that drive sustainment costs. Size and complexity of the software is important. The more software you have the more bugs you are likely to have and the more technical debt you are likely to have incurred. Larger applications also require that more time and effort be devoted to acceptance and regression testing. Add complex functionality to the mix and it's likely to take more time and effort to fix bugs, add features, refactor code and make necessary upgrades to the technology. Other important factors in assessing sustainment costs include the technology used to develop the software system, team experience, tool sophistication and the age of the application.

There have been many approaches suggested in the literature for determining the maintenance costs for software. Boehm proposes a modified version of the COCOMO model with maintenance size driven by maintenance change traffic, software understanding (a proxy for the quality of the architecture and code) and programming familiarity [2]. Bachmann proposes a model that is driven by software size, complexity, components, interfaces, change traffic, experience and code quality.[3] Capra suggests that entropy, representing the factors that cause software to degrade over time (encompassing structural quality) is an important factor in determining maintenance costs for software. [4] Other models exist as well in the literature for estimating maintenance costs to varying degrees of detail. [5] [6] [7]

Software Quality

When referring to software, there are really two dimensions to quality; functional quality and structural quality. Functional quality indicates how well the software meets the functional requirements; how well does it satisfy the end user or businesses requirements. Structural quality reflects how well the software was produced. Clearly, poor structural quality can lead to the appearance of poor functional quality; if the application crashes because of bad architectural decisions, the user can't get to the required functionality. While the two types of quality are intermingled in the mind of the end user, they have significantly different implications to the software sustainment community. The structural quality of a software application can significantly impact software maintenance costs.

The Consortium for IT Software Quality (CISQ) is currently working to develop structural quality standards for software. CISQ has identified 5 desirable characteristics that a software application must possess to be considered a quality solution that delivers real business value. Of these five, four are related specifically to the quality of the code.[8] The characteristics are:

- Reliability – this indicates resiliency and structural solidity. This provides a measure of the likelihood of potential application failure as well as giving an indication of stability in the sense of how well the software will hold up to modifications.
- Performance – the architecture and design of the software supports high performance operations in the intended environments.
- Security – the architecture and code are well constructed to ensure that there are no security breaches
- Maintainability – the architecture and code support effective changeability, adaptability, portability, and the transfer of code from one development team to another.
- Size – while not really a quality metric on its own, size contributes to maintainability.

While it is true that these characteristics indicate a quality product it's not at all clear how to quantify them for an existing software program. CISQ is currently in the process of creating a standard for quantification of these characteristics based on characteristics of the code base but this standard is still in the works. This being said, it's not rocket science to understand what characteristics of code would lead to higher or lower reliability (or efficiency, security, etc).

Reliable, efficient, software has a well thought out, easy to understand and well documented architecture. Complexity is minimized in the design, code and interfaces. Error handling and exception handling are pervasive throughout the code. Programming best practices and sound resource management techniques are applied.

Structurally secure software also requires good architectural decisions but mostly it needs to be well written. It is free of back doors, buffer overflows or other defects that allow entry to hackers. The Software Engineering Institute reported in 2005 that 90% of software vulnerabilities were a result of defects in the software. [11] Security breaches in software are caused by defective specification, design and implementation.[9] Buffer overflows, which offer hackers entrée into the system continue to be cited as one of the most common software errors [10]. Other defects that lead to security breaches include format bugs, hardcoded path names and passwords and malformed inputs.

Maintainable software is well written, well documented and follows programming best practices. It is simplistic and does not include functionality that is not needed. The code is well commented and organized in a logical fashion with lots of modularity so that it can be easily transferred to new programmers or teams. The development environment is also logically structured with sensible file structures and enforced naming conventions. Maintainable software includes a suite of acceptance and

regression tests that provides complete coverage, making it easy to make changes without worrying about breaking the existing functionality.

Quality Best Practices

Truly the best way to achieve high quality code is to invest in good practices during development. There are many proactive ways to accomplish this. All of these require some investment during development but that investment has been proven to pay off as the software is deployed and transitions from development to sustainment budgets.

Pair Programming or Peer Reviews

Pair programming and peer reviews are often cited as leading to improved software quality. With pair programming, two programmers are paired together to complete a single programming task. They have one computer between them and they collaborate to determine the best design and best implementation for that design. Pair programming applies the "two heads are better than one" paradigm while enforcing on-going continuous review of the code.

Peer reviews are generally a more formal and focused review of artifacts. These reviews can be focused on requirements, architecture, design, code and/or test plans. The nature of the project (size, complexity, and criticality) determines the scope of peer reviews, their frequency and the comprehensiveness of review coverage.

While the results are mixed on whether pair program and peer reviews increase or decrease productivity, impacts on quality have been documented in many studies. In a study reported on in [12], comparing programs developed by single programmers to those developed by pairs, the number of post development tests that passed increased by 15%. The report also presented experiential data that pair programming increased the quality of designs as well. [13] Cites two examples supporting the quality benefits of pair programming – an experiment conducted at the University of Utah finding an average of approximately 14% increase in passed tests on programs using pair program and experiential findings on the Chrysler Company's C3 project in 1997. Capers Jones book *Software Engineering Best Practices [14]* rates peer reviews and inspections as the top practice (or practices) for defect removal for projects of all sizes.

Test Driven Development

Test driven development requires that no code is written for a feature until the tests for that feature have been written and shown to fail. Before a developer can begin to code for a particular feature they need to understand the requirements, intent and exceptions of the feature well enough to write tests for it. The process for each new feature begins with a study of the user story in order to assess requirements and write an automated test for the feature. This test is then executed to ensure that it

fails (since the feature has yet to be written). Once the test has been proven to fail, the developer writes what they believe to be the minimal amount of code to make the test pass. Once enough code has been written to make the test pass, the developer then incrementally refactors the solution to make it simpler and cleaner. Because each increment includes rerunning of the test, the developer can refactor with confidence that the feature will not be negatively impacted.

Tests conducted at Microsoft in two different environments showed defect rate (defects/KLOC) decreases at factors of 2.5x and 4.2x between projects determined to be similar in size and scope, one with TDD and one without.[14] Both tests indicated increased development time as well but not nearly as significant as the increase in quality. [15] contains two tables which summarize the results of 18 studies – 9 from academia and 9 from industry. These studies ranged from controlled experiments, to quasi-controlled experiments to case studies and the length of studies ranged from 1.75 hours to 1.5 years. Different measures of quality were employed by different studies (functional tests passed, defect rates) Of the studies, 10 found mild to significant improvement in quality, 7 were inconclusive or showed no difference while only one of the studies indicated a decrease in quality. Lisa Crispin in [16] correctly points out that there is more to quality than defect counts or passed functional tests. Her teams employ customer test driven development in addition to developer test driven development. Testers work with customers to develop high level functional tests for features. The developers use these tests to fully understand the user's requirements. The result is happy (often delighted) customers – contributing to the 'doing the right thing' aspect of quality as well as the doing it right.

Continuous integration with automated tests

Continuous Integration is practice that requires that changes to the code base be continuously (or regularly within short periods of time) integrated into an operational system. Generally this process is automated and integrated with an automated test suite in order to give real time feedback when a developer makes a change that causes bad behavior in unexpected places in the system. During development, individual developers will 'check out' a copy of the code, make changes to fix a bug or add a feature, test this change and then 'check in' the changed code. During this time, other developers have made changes to other parts of the code base. A common problem arises when the 'check out' time is too long because while each individual developer is able to pass the suite of tests with their instance of the code, the combination of their code with the changes of other developers causes tests to fail.

Continuous integration requires that an application is rebuilt and tested each time a change is made to the code base. Some shops automate this process through their version control system while others have automatic builds and tests that run every couple of hours with developers checking in code often. Solving any failures in these tests should become a top priority for the team because failed tests are much cheaper to fix when the changes are fresh in the minds of the development team. When there are failures found during integration it is easy to isolate the problem code and correct the problem. The longer the period between integrations the more likely the team is thrown into integration hell – unable to determine which of many code changes is responsible for failed tests.

While there doesn't appear to be much quantitative evidence specifically relating continuous integration to increases in quality, it has been observed that projects using continuous integration tend to have dramatically less bugs in production and in process[17]. Steve McConnell in [18] indicates that one of the benefits of frequent builds is reduced risk of low quality. Clearly this is influenced significantly by the number and quality of the tests and the amount of automation applied to the testing for each build. Quality impacts also depend significantly on how seriously the team acts when tests do fail. If failed tests are not treated as a top priority the value of continuous integration is seriously depleted.

There is an upfront investment in both hardware and software to create an environment for continuous integration as well as the on-going maintenance required to keep tests up to date as the software system evolves. Despite this many find that the overall impact on the cost of development is decreased with continuous integration because machines are being used to free up human time for more productive endeavors than rote testing. [21] Other studies indicate that continuous integration increases project costs. It appears that the size of the project is a big factor in the productivity impacts of continuous integration. On large projects or with widely distributed teams it seems to have a definite positive impact on productivity while with smaller, well contained projects it's less clear of the cost benefits. The quality benefits apply to projects of all sizes.

Automated Static Code Analysis

A static code analyzer is an automated tool that reviews software code to detect errors in the software as well as to determine the structural soundness of the code. Static analysis is an automated code review that uses patterns, best practices and standards to identify violations in software code. These violations may or may not lead to defects in the executed code, but there presence in the code indicates an area of potential weakness or a function that may be difficult to maintain in the future. Static analysis tools also collect metrics about the size and quality of the software code.

Static code analysis can be performed at any point in the project. Errors in code can be detected even before the code is in an executable state. Defects detected early on are generally much less expensive to fix than those identified later in the project. Static analysis clearly leads to higher quality code, it has been shown to reduce defects by as much as a factor of 6.[22] Capers Jones cites the use of static analysis tools as a best practice for defect removal, second only to inspections and peer reviews. [4] More and more organizations are seeing the value in static analysis, especially those building software for medical equipment or with high security requirements.

Static code analysis is not cheap. There are open source tools available but in order to be effective, they need to be configured to conform to an organization's specific best practices and standards. There is an upfront investment to learn to use the tool as well as on-going requirement to spend time maintaining the configurations and analyzing the results that emerge from the tool. When properly implemented and effectively used, static code analysis has been shown to save organizations time and money by finding and fixing defects early in the project lifecycle and by helping create more maintainable software. While the costs of tools, implementation and training may not be justified for small projects

or projects with low security and reliability requirements, there are many instances where the increased quality and reduced maintenance cost more than make up for the increased cost during development.

Quality documentation

According to the SWEBOK, software engineers spend 40-60 % of the maintenance effort determining where to make a change or correction [2]. A study in [19] determined that source code and comments are the most important artifacts when it comes to understanding software systems. An experiment conducted in [20] found that software engineers who had good documentation spent 21.5% less time understanding the software that was to be changed than those who had only source code.

Even without the references cited above, it seems to be incredibly intuitive that if the maintenance team has a clear understanding of the requirements, architecture, design, code, test plans and other project factors they will be more efficient, both in finding and fixing bugs but also for changes intended to enhance functionality, improve performance or refactor code. Certainly well written project documents such as Requirements, Specification and Architecture are a vital roadmap to the decisions that were made during the development process. Similarly a good test plan, hopefully accompanied by a comprehensive set of tests, will ensure the maintenance team isn't breaking existing functionality as they add new. And while many developers feel that their code is self explanatory when they write it, most will admit that even they sometimes have a hard time understanding what their train of thought was when they revisit it after time has passed. Code replete with informative comments is easier to maintain than code that isn't.

Naming conventions

While there is little in the literature that makes a link directly between good naming conventions and costs of software maintenance, there is evidence that the time it takes to understand code can be a significant part of the overall maintenance effort. Examine the following two coding options for calculating the area of a square:

```
x = y * z;                          (1)

SquareArea = height * weight;       (2)
```

It's not rocket science that a new developer (or even the original developer given some time away from the code) will understand the intent of the code in (2) more quickly than she will that in (1). Back in the day, when programming languages limited variable names to six or eight characters, software developers were significantly challenged to create meaningful variable names but this is no longer the case. Development teams should create and enforce a strict naming convention that is as descriptive as the language allows so that the code reads as much like a novel (albeit most likely a boring novel) as possible.

In addition to being aware of naming conventions used in the code, software development teams should also be aware that the file structure and file naming conventions are also an important consideration when thinking about the long term maintainability of a software system. Before the maintenance engineer can begin to understand a piece of code that is to be corrected or changed, they need to be able to find that piece of code. If the development team has developed, documented and enforced rules for file naming and file structure, this will act as a roadmap for the maintenance engineers that will simplify the task of finding specific parts of the system.

Conclusions and Recommendations

For most software applications, the amount of money spent sustaining it is more than the amount of money that goes into developing it. Sustainment consists of many activities including fixing defects, adding new features, modifying the software to work in changing environments, supporting the software in the field and addressing technical debt. The cost of all of these activities is a function of the quality of the software. Clearly maintenance costs decrease as the quality of the code increases, but other costs decrease as well. Enhancements are easier to make in software that has high quality and maintainability. Higher quality software has less technical debt and is easier to refactor.

Quality comes in two flavors. Software that does that right thing has high functional quality. Software that does the right thing the right way has high structural quality. Structural quality is determined by assessing the software for reliability, performance, security and maintainability. There are many practices which can lead to high structural quality. Some of these come at virtually no cost to the development effort. Practices such as good naming conventions, well commented code, and test driven development. Some practices may come at a cost such as pair programming, peer reviews and continuous integration practices so the development team needs to make an evaluation of potential costs and benefits based on the type of software being developed and the nature of the developing team and organization. Some practices, such as static code analysis and increased documentation will add costs to the development process but have been shown to increase significantly the quality and maintainability of the software.

Every project is different. All projects should invest in some sort of code reviews, inspections or pairing opportunities. This alone has significant impact on project quality and the cost can be contained by limiting the scope to the size and nature of the project. Small projects are likely to get the best benefit from practices like pair programming, test driven development and well enforced policies for commenting code and naming conventions. Medium size programs would benefit from code inspections, continuous integration and some level of static analysis. The larger a project gets the more important it is to have various levels of peer reviews or inspections, not just code but requirements, architecture, etc. They should also consider making a serious investment in static code analysis particularly if there are reasons why failure of the software is not an option.

There is no easy answer. Best practices, good documentation and good naming conventions are essential to successfully developing, delivering and maintaining high quality software. No team should spend much time with a cost benefit analysis around these practices. There are other practices a team

may consider adopting to increase the quality of the software they deliver. Which of these practices makes the most sense for a team depends on the size of the software project, the nature if the team and the organization and the functionality and requirements for the software being developed. Some of these practices will decrease project productivity during development while others have been shown to increase development productivity. It is important that teams understand that any new practice, even one touted as a productivity or quality enhancer, may not show improvements right out of the gate. Organizations should gradually introduce improvements to their process and allow time for them to be embraced before assessing the success of the practice in their organization.

References

[1] Boehm, B. W., *Software Engineering Economics,* Prentice-Hall Inc., Englewood Cliffs, NJ, 1981
[2]Software Engineering Book of Knowledge available at http://www.computer.org/portal/web/swebok/html/contents (retrieved March 2012)
[3] Buchmann, Irene, Frischbier, Sebastian, Putz, Dieter ,"Towards an Estimation Model for Software Maintenance Costs", 2011 1th European Conference on Software Maintenance and Reengineering. IEE Computer society pp313-
[4] Eugenia Capra, Chiara Francalanci, Francesc Merlo, "The Economics of Open Source Software: An Empirical Analysis of Maintenance Cost, ICSM 2007
[5] Sneed,, Harry M. "A Cost Model for Software Maintenance & Evolution", Proceedings of the 20th IEEE International Conference on Software Maintenance, 2004
[6] Hayes, J.H., "A Metrics Based Software Maintenance Model", available at http://selab.netlab.uky.edu/homepage/hayesj_sw_maintenance_effort_csmr_04-Revised.pdf (retrieved March 2012)
[7] "Yu, L., "Indirectly predicting the maintenance effort of open-source software", Journal of Software Maintenance and Evolution: Research and Practice", 2006, 18:311-332
[8] OMG, SEI, "CISQ Executive Forums Meeting Minutes and Next Steps", Dec 2009, available at http://it-cisq.org/images/stories/cisq_director_executive_report.pdf (retrieved March 2012)
[9] Software Process Subgroup of the Task Force on Security Across the Software Development Life Cycle, "Process to Produce Secure Software", National Cyber Security Summit, Mar, 2004
[10] Common Weakness Enumeration, "2011 CWE/SANS Top 25 Most Dangerous Software Errors", available at http://cwe.mitre.org/top25/ (retrieved March 2012)
[11] Davis, Noopur;" Secure Software Development Life Cycle Processes : A Technology Scouting Report", December 2005, CMU/SEI-2005-TN-024
[12] Cockburn, A,, et.al., "The Costs and Benefits of Pair Programming", available at http://collaboration.csc.ncsu.edu/laurie/Papers/XPSardinia.PDF (retrieved Jan 2010)
[13] Williams, L., et. al., "Strengthening the Case for Pair-Programming", available at http://www.cs.utah.edu/~lwilliam/Papers/ieeeSoftware.PDF (retrieved Jan 2010)
[14] Bhat,T, et.al, "Evaluating the Efficacy of Test-Driven Development: Industrial Case Studies", ISESE 2006, Sept 21-22, 2006, Rio de Janiero, Brazil, ACM 1-59593-218-6/06/2009
[15] Jeffries,R., et. Al. "TDD: The Art of Fearless Programming", IEEE Software, May/June 2007, pp24-30
[16] Crispin, Lisa, "Driving Software Quality: How Test-Driven Development Impacts Software Quality, IEEE Software, November/December 2006, pp70-71
[17] Fowler, Martin, "Continuous Integration", http://www.martinfowler.com/articles/continuousIntegration.html#BenefitsOfContinuousIntegration (retrieved Jan 2010)
[18] McConnell, Steve, "Daily Build and Smoke Test", IEEE Software Vol 13, No 4, July 1996
[19] de Souza, Sergio Cozzetti, et.al., "A Study of the Documentation Essential to Software Maintenance", ACM SIG for Design of Communication, Proceedings of the 23rd annual International Conference on Design of Communication, 2005
[20] Tryggeseth, Eirik; "Report from an Experiment: Impact of Documentation on Maintenance", *Empirical Software Engineering, Vol 2(2), 1997, pp201-207.*
*[*21] Krill, Paul; "Why the time is now for continuous integration in app development",. InfoWorld, July 7, 2011 available at http://www.infoworld.com/d/application-development/why-the-time-now-continuous-integration-in-app-development-941?page=0,1 (retrieved March 2012)
[22] Xiao, S., et. al, "Performing high efficiency source code static analysis with intelligent extensions", Proceedings APSEC, 2004

