Using Software Quality Methods to Reduce Cost and Prevent Defects

Rick Spiewak
The MITRE Corporation

Karen McRitchie
Galorath Incorporated

Everyone knows that it's better to "do it right the first time." But in organizations, this requires the ability to predict outcomes of their established "best practices" as well as the ability to justify costs when it comes to applying what may be new approaches. This is just as true in software development as it is in any other business practice. This article will survey some of these best practices and present a method for evaluating the costs and benefits of applying them.

Software can be considered a product whose production is fundamentally similar to other products. Improving the quality of software can be approached using the same basic principles espoused by quality pioneers such as W. Edwards Deming, Philip B. Crosby, and Harold F. Dodge. These principles can form a practical framework for ensuring that appropriate requirements are set for software development projects. By connecting established software engineering practices to the objective of defect prevention, we can apply the principles of quality management to software development. Using modeling techniques, it is possible to predict the potential cost savings and defect reduction expected.

Quality management is a well-established discipline with historic roots in manufacturing industries. W. Edwards Deming [1], Philip B. Crosby [2], and others have written and taught extensively in the field. The classical approach to quality management can be summarized in these simple steps:

1. Analyze product defects to determine root causes.
2. Modify processes to address and remove root causes of defects.
3. Fix defects using improved processes.

By following this approach, we can realize the goal of improving product quality by removing the causes of defects. As Crosby put it: "Quality is free. It's not a gift, but it is free. What costs money are the non-quality things—all the actions that involve not doing jobs right the first time" [2].

Identifying Best Practices

While the classical approach to quality management (as taught by Crosby and others) would suggest that each organization should start fresh in identifying and fixing process defects in order to improve product quality, experience suggests otherwise. The famous mathematician and computer scientist Richard Hamming once asked, "How do I obey Newton's rule? He said, 'If I have seen further than others, it is because I've stood on the shoulders of giants.' These days we stand on each other's feet" [3].

If we want to profit from the work of pioneers in the field of software quality, we owe it to ourselves and them to stand on their shoulders. This means that we should be willing to adopt proven best practices without necessarily requiring that their value be proven first in our specific development environment. We will try to identify some of these best practices and focus our attention on them here.

Narrowing Our Focus

There is no doubt that the quality of software is heavily influenced by proper attention to every phase of development, from conceptual design through requirements definition, architecture, detailed design, construction, testing, documentation, training, deployment, and sustainment. However, for the purposes of this article, we will focus on what Steve McConnell refers to as the *software construction* [4] phase of software development.

Best Practices in Software Construction

This article will treat four areas of best practices in software construction:

- Uniform coding standards.
- Automated Unit Testing (AUT).
- Root cause analysis.
- Code reuse.

These areas, in turn, can be linked to the four software construction fundamentals cited in the IEEE Computer Society's "Guide to the Software Engineering Body of Knowledge" [5]. It stated that the fundamentals of software construction include:

- Minimizing complexity.
- Anticipating change.
- Constructing for verification.
- Standards in construction.

Proper attention to these areas will lead to improved quality in the software we create, while moving closer to Crosby's idea that quality can, in fact, be *free*.

Uniform Coding Standards

Coding standards incorporate experience and best practices at a detailed level into the software construction process. Typically, these include the seemingly trivial, such as spelling, use of names, and upper/lower case; the moderate, such as the code matching the in-line documentation; and the critical, such as proper management and disposition of objects, exception handling, completeness of branch tests, and so forth.

The use of uniform standards provides a wide range of benefits:

- **Readability.** Any programmers writing to the same standards will be better able to read, critique, or even take over software written by others. This saves time and avoids misunderstandings in areas including peer reviews, updates and maintenance, and reassignments.
- **Support by tools.** Static analysis tools are available for contemporary programming languages and environments, which incorporate the ability to check for adherence to coding standards and best practices in writing code. By adopting the standards supported by these tools, we obtain the advantage of increased automated tool use, one of the metrics used in the System Evaluation and Estimation of Resources – Software Estimating Model (SEER-SEM), as referenced in the forthcoming Using the Model section.
- **Peer review benefits.** The peer review process is enhanced by the adherence to uniform coding standards. Code is more accessible to potential reviewers and less time is wasted adapting to differing approaches. The review can focus on actual and potential defects and their causes.

In addition to checking for adherence to standards, peer review leads to the sharing of ideas and improved coding techniques. Inspection by the developer prior to review may contribute to defect prevention as well.

Government audit of the peer review

process is enabled by coding standards. Software should be randomly reviewed on behalf of the customer in order to ensure that a uniform approach is being followed.

AUT

Developers have long been responsible for unit testing their code. This involves testing the smallest possible part of a program to ensure correct operation. Techniques for unit testing include the use of a debugger to step through a routine, following a script which exercises the desired functions, and the use of a test *harness* or *framework* to execute tests automatically. The last of these techniques is generally referred to as AUT.

While strategies for analyzing unit test requirements include Dr. Thomas Radi's TestGen for Ada [6] from 1989, the best-known lineage of modern AUT dates from SUnit for Smalltalk [7] in 1994. This was followed in 1999 by JUnit (for Java). Because of this heritage, and the fact that the basic structure of these test harnesses has been carried forward into multiple environments, AUT is often represented by the *XUnit* family of test harnesses, including JUnit and NUnit (for Microsoft.NET). Note that the *automation* in this family of test harnesses is in the *execution* of tests. There are other tools available that will help to create at least a skeleton of the tests themselves. It is up to the developer to ensure, by the use of tools and inspection, that there is sufficient coverage of input values and paths through the code to provide the desired thoroughness.

There is a positive impact on design when automated unit tests are implemented from the beginning. In order to prepare for AUT, code must be designed to be tested. Construction techniques such as *dependency injection* or *dependency lookup* [8] help to reduce coupling between software modules, enhance modularity, and aid in testability. For a more complete discussion of unit testing patterns and techniques, see [8].

In addition to aiding in the initial assurance of correct operation, automated unit tests serve as regression tests for existing methods and routines in the course of development by continuing to test previously working code each time they are run. During maintenance and enhancement, this regression testing helps to prevent the introduction of new errors into existing code.

When used together with requirements for code coverage, automated unit tests can be used to both prevent regression errors and set minimum standards of correct operation. There are tools available which will work in conjunction with unit test tools to show how much of the code under test is being executed in a particular scenario.

One well-known approach to AUT, Test Driven Development [9], extends the testing model so that tests are written first, then code is added to pass the tests. This has the additional benefit of focusing development on the requirements and discouraging what has been called *feature creep*: adding features or capabilities while programming.

The "Haves" and the "Have-Nots"

What we found, in an informal survey of users of AUT, is that development organizations which use it do not have detailed cost comparisons available. In general, once they started using it, they just never went back to traditional manual methods, nor have they deemed it worthwhile to conduct comparative studies. Those who have not yet adopted these tools have sometimes not done so because of the perception that it will cost more. We will attempt to show that, over the course of development, this perception is false.

"When used together with requirements for code coverage, automated unit tests can be used to both prevent regression errors and set minimum standards of correct operation."

Root Cause Analysis

Root cause analysis is the most fundamental technique of quality management, and is a CMMI Level 5 practice area. It is important to use this technique, however, regardless of the CMMI level. Fixing defects in a product without addressing the cause is known in classic manufacturing environments as *rework*. It is no different in software development, where we call it *fixing bugs*. Without addressing root causes, there is no reason to believe that simply reworking software defects will improve the quality of the overall result, since the same (potentially flawed) processes are used to make the changes as were used originally to write the code.

Accepted techniques for analyzing root causes include the *5 whys* method and Kepner-Tregoe Problem Analysis method [10]. The former method was originally developed at Toyota Motor Corporation and is deceptively simple: When analyzing a defect or failure, start by asking *why?*, and continue asking this for each answer until a satisfactory root cause is reached. The number 5 is simply a guideline in this case.

Kepner-Tregoe's Problem Analysis takes a different approach, asking (with regard to a defect or failure): *What? Where? When?* and *To What Extent?* These questions are addressed in terms of what the problem is, what it could be (but is not), and what changes and differences are associated with the occurrence. These are then analyzed for determining possible root causes.

Analyzing and addressing root causes is essential to improving the development process. However, in order to preserve and then later analyze the knowledge gained by this approach, it is necessary to classify root causes.

Classification: Root Cause Taxonomy

A variety of schemes have been proposed and used for classification of root causes. These include IEEE Standard 1044 and IBM's Orthogonal Defect Classification [11]. However, these do not lend themselves well to automated analysis.

Boris Beizer [12] provided a simple approach to root cause classification. The Beizer Taxonomy yields a 4-digit number. Based on an open-ended hierarchical classification scheme, it can be extended without changing the original categories.

One of the advantages of this approach is that it is amenable to analysis using database query tools, Pareto diagrams, and statistical techniques for extracting patterns from data. Consistent use of this taxonomy can provide an enterprise with insights into areas for process improvement that might not be readily apparent otherwise. The enterprise, for example, may be a customer for software which is written by a variety of development organizations.

Table 1 shows the top level categories of the Beizer Taxonomy.

Software Reuse

Through the use of uniform coding standards and designing software to be readily tested by automated unit tests, there is an increased likelihood that software devel-

oped for one project or program can be reused by another. Good documentation and modular design are also needed to make software reusable.

Additional value can be added when reliable open-source or other freely available software with a large body of users can be applied to a project. Depending on the development environment and language(s) involved, this may mean looking at open-source projects or libraries such as the Microsoft Enterprise Library (for the .NET Framework).

Many of these sources meet the other previously discussed criteria, such as using well-established coding standards and including automated unit tests. Available software which does not meet these criteria has an implicit cost in adopting it for reuse: Bringing the software up to the same standards used during specific development for the project may be required.

Cost/Benefit Analysis

In order to analyze the benefits of introducing techniques aimed at improving software quality, we need to find a way to predict the results. This is accomplished through the use of modeling and comparing the predicted outcome of the development effort under varying conditions and practices.

Modeling the Cost of Improving Quality

Crosby defines the *cost of quality* as "… the expense of non-conformance—the cost of doing things wrong" [2]. This can be added up after the fact as the cost of rework and scrapping the work (in the case of software, the cost of fixing defects in the code). But, suppose we want to predict ahead of time what the benefits might be of applying some of the fundamental best practices previously described to a development project? Take, for example, the problem of the benefits of AUT and static analysis.

Costs and Benefits of AUT

One of the most intractable problems in considering the introduction of best practices into the software construction phase has been justifying the cost. If you attempt to look at the value of AUT in isolation, this fundamental problem presents itself: Most sources agree that to test *n* lines of source code requires at least *n* to *n + 25* percent additional lines of code [13]. If you apply this to a traditional estimating approach which multiplies source lines of code (SLOC) by hours or dollars, it will appear that the use of this technique will add significant cost to the project. If this were so, then the apparent fact that organizations using this technique are so thoroughly committed to it would seem to be contradictory.

Fortunately, there is a more comprehensive approach. The cost modeling software which is in use by the AF Electronic Systems Center at Hanscom AFB takes in to account a number of factors in addition to SLOC. These include the use of automated tools, the degree of testing, and the extent of quality assurance. This allows us to see past the SLOC issue, and estimate the savings which are possible by the use of these techniques.

Using the Model

The commercially available SEER for Software application is a comprehensive software project estimation system. SEER-SEM is the core estimation capability originally based on the effort and schedule relationships developed by Dr. Randall Jensen [14]. The SEER-SEM product comes with a comprehensive set of knowledge bases which offer default parameter values that target complexity and productivity factors for a wide variety of project types. These knowledge bases are developed and tested by analyzing thousands of projects. Initial inputs to a SEER-SEM estimate include a description of the platform, application type, reuse scenario, development methods, and development standards. Detailed inputs include several ways to enter software project size as well as several productivity-related parameters that help describe the people developing the software, the methods and tools used, the customer-driven requirements and constraints, and the system being developed. This allows the user to do *what-if?* analysis based on a variety of development strategies using various parameters related to the size of a project, its difficulty, the experience of the developers, and the tools and techniques used.

The use of a cost modeling tool to do *what-if?* studies serves as a means to simulate different scenarios. Ideally, it can provide an objective assessment of how cost, schedule, and quality might change as project assumptions change. In using SEER-SEM, you can evaluate the impacts of project assumptions to the whole project, not just the construction phase that is most directly impacted by AUT and static analysis tools.

From the perspective of cost modeling, AUT, along with tools that check source code for syntax or security errors, fall into the general category of *automated tool use*. According to the SEER-SEM model [15], increasing the use of automated tools actually decreases (rather than increases) the cost of developing software. In addition, it reduces the number of defects expected to be produced by the process.

Top-level categories:
• 0xxx Planning
• 1xxx Requirements and Features
• 2xxx Functionality as Implemented
• 3xxx Structural Bugs
• 4xxx Data
• 5xxx Implementation
• 6xxx Integration
• 7xxx Real-Time and Operating System
• 8xxx Test Definition or Execution Bugs
• 9xxx Other

Table 1: *Top-Level Categories of the Beizer Taxonomy*

In one example, changing the model parameter *Automated Tool Use* from Nominal to High, resulted in a projected decrease in effort of 9 percent, accompanied by a decrease of 13 percent in predicted defects. This shows that, contrary to a cursory estimate, doing the extra work to develop automated unit tests (along with other automated tool use) can be expected to reduce the overall effort involved in software development. While this result appears interesting, it is important to understand that changing a single parameter to study the cost and quality trade-off of AUT can be viewed as overly simplistic. Fortunately, there are other dimensions to this scenario that can be evaluated using a cost modeling tool.

It is fair to say that introducing automated tool use into a development organization will not produce instant benefits. Fortunately, the cost modeling tools allow for a more nuanced look at this *what-if?* scenario. In addition to looking at the impact of automated tool use improvement, we can consider experience factors and the potential for added volatility.

As an example, we will examine a project with three major applications and two vendor-supplied applications. The project is of moderate criticality in terms of the overall specification, quality assurance, and test levels required. There are three cases examined:

- **Baseline:** Assumes no AUT, which notionally represents the organization *as-is*. The team has nominal experience with the development environment, tools, and practices.
- **Introducing AUT:** Takes the baseline scenario with the introduction of

	Baseline	Introducing AUT	Difference	AUT + Experience	Difference
Schedule Months	17.09	17.41	2%	16.43	-4%
Effort Months	157	166	6%	139	-11%
Hours	23,881	25,250	6%	21,181	-11%
Base Year Cost	$2,733,755	$2,890,449	6%	$2,424,699	-11%
Defect Prediction	84	81	-4%	68	-19%

Table 2: *Cost Model Trades*

AUT. This will result in a small increase of the automated tool use parameter as well as the modern development practices. However, since the use of these tools is new to the organization and teams, there is a decrease in the overall development environment experience. Also, introduction of new tools may inject some volatility into the system. This is because the team may need to tweak the process to accommodate the tools being used. For example, they may need to upgrade to the latest service packs for their operating system or development environment in order to integrate the unit test tools effectively.

- **Introducing AUT and Added Experience:** Similar to the previous, but with the caveat that the team has had some training in the use of the AUT tools and has established the methods used as *routine*. This training may be done in a traditional classroom or *boot camp* environment, or it could be on-the-job training. In either case, the assumption here is that the team has gained some experience in using the AUT tools and the process is well integrated into their overall development process.

The results of these three runs are shown in Table 2.

Results include the estimated schedule months or duration and the estimated effort expressed as effort months, hours, and cost. The last row in the table is the estimated number of delivered defects. The defect estimate in SEER-SEM takes into account the project size, programming language used, as well as many of the productivity factors used to estimate effort (e.g., requirements definition formality, specification level, test level, and others).

The results of this analysis demonstrate that there is an initial hit to overall productivity when introducing new tools and methods. However, this impact is not a long-term change, but rather a short-term setback that can be overcome by training or general experience. It is worth noting that even without the benefit of experience, the number of defects went down with the introduction of AUT.

By adding the dimension of defect prediction into the cost-modeling method, you can quantify the impact of changes in tools, methods, and staff capabilities used on a project, not just in terms of investment or savings, but also in terms of improved quality. Software managers need to be able to justify investments in new tools and technologies, but using claims by tool vendors can be misleading. Investment in quality improvements should be analyzed, not just for the general effects, but for their effects on specific projects. It is important to not just look at the benefit of the coding effort (as many tool vendors will provide), but to the overall benefit of the project.

In addition to the end result, visibility into the defect profile over the development period is available as part of this cost model. The *defect prediction* is considered to be defects delivered at the end of development. However, projects find and remove many more defects during development. Every project has a *defect potential* that represents the opportunity for defects to occur during development and beyond. The defect potential is based on size, complexity, and other factors. In general, most of the potential defects are found and removed through the development process. However, not all are removed, leaving delivered defects. The percentage of defects removed during development is called the *defect removal efficiency*. This is calculated as the total defects removed divided by total defect potential. Higher defect removal efficiencies are typically associated with the use of more rigorous or formalized software development methods.

The detail behind the quality metrics in this analysis, shown in Table 3, is provided by the cost model. When introducing AUT, you see a small increase in the defect removal efficiency. However, this increase is offset by an increase in the overall defect potential that results in an increased number of hours spent removing each defect. However, when you couple AUT with the requisite experience, the increase in defect removal efficiency is boosted by the fact that the overall defect potential is reduced. This reduction in defect potential, combined with the overall effort reduction, quantifies the intuitive adage that the cheapest defect to remove is an avoided defect.

While cost modeling tools have been used for budgeting and proposal purposes, they can be employed as a strategic tool to evaluate how changes in processes and methods will impact a software development organization. Cost modeling tools provide a tangible method for understanding how the use of new methods and tools can impact cost, schedule, and quality. In this case, it was demonstrated that investment in quality methods is justified. Additional benefits can be obtained when looking at required maintenance efforts. Having fewer defects means that less time is spent fixing problems, giving more time and resources to improving the system.

Table 3: *Defect Prediction Detail*

	Baseline	Introducing AUT	Difference	AUT + Experience	Difference
Potential Defects	738	756	2%	668	-9%
Defects Removed	654	675	3%	600	-8%
Delivered Defects	84	81	-4%	68	-19%
Defect Removal Efficiency	88.6%	89.3%		89.8%	
Hours/Defect Removed	36.52	37.41	2%	35.30	-3%

Summary

Adopting and enforcing best practices in software construction leads to better results at a lower cost. The practices outlined in this article are a good starting point for a quality improvement program in the construction phase of software development. These best practices can be implemented directly by a development organization, or incorporated into contractual requirements by an acquisition organization. Modeling tools can be used

to demonstrate the cost effectiveness of implementing best practices, and to help justify any initial cost to the development organization in instituting these practices.◆

References

1. Deming, W. Edwards. Out of the Crisis. MIT Press, 1982.
2. Crosby, Philip B. Quality Is Free: The Art of Making Quality Certain. McGraw-Hill Companies, 1979.
3. Hamming, Richard. You and Your Research. Transcription of the Bell Research Colloquium Seminar, 7 Mar. 1986.
4. McConnell, Steve. Code Complete 2. Microsoft Press, 2004.
5. IEEE Computer Society. Guide to the Software Engineering Body of Knowledge. 2004 Version.
6. Radi, Thomas. TestGen – Testing Tool for Ada Designs and Ada Code. Proc. of the Sixth National Conference on Ada Technology, 1989.
7. Beck, Kent. "Simple Smalltalk Testing: With Patterns." XProgramming.com. Oct. 1994 <www.xprogramming.com/testfram.htm>.
8. Meszaros, Gerard. xUnit Test Patterns: Refactoring Test Code. Addison Wesley Professional, 2007.
9. Beck, Kent. Test Driven Development: By Example. Addison-Wesley, Nov. 2002.
10. Kepner, Charles H., and Benjamin B. Tregoe. The New Rational Manager. Kepner-Tregoe, 2006.
11. Chillarege, Ram, et. al. "Orthogonal Defect Classification – A Concept for In-Process Measurements." IEEE Transactions on Software Engineering. Vol 18, No. 11, Nov. 1992.
12. Beizer, Boris. Software Testing Techniques. 2nd ed. International Thomson Computer Press, 1990.
13. Conversation with Dana Khoyi, Vice President and Chief Technical Officer of Global 360, Inc.
14. Fischman, Lee, et al. "Inside SEER-SEM" CrossTalk Apr. 2005.
15. Galorath Incorporated. SEER-SEM User Manual. Aug. 2006.

About the Authors

Rick Spiewak is a lead software systems engineer at The MITRE Corporation. He works at the Electronic Systems Center at Hanscom AFB as part of the 951st Electronic Systems Group, concentrating on Mission Planning. Spiewak's current focus is on the software quality improvement process and he spoke on this topic at the 2007 AF Information Technology Conference. He has been in the computer software industry for more than 37 years, and has bachelor's and master's degrees in electrical engineering from Cornell University. He also studied quality management at Philip Crosby Associates.

The MITRE Corporation
Command and Control Center
MS 1614E
202 Burlington RD
Bedford, MA 01730
Phone: (781) 266-9672
Fax: (781) 266-9748
E-mail: rspiewak@mitre.org

Karen McRitchie is vice president of development at Galorath Incorporated, and is responsible for design, development, and validation of current and new SEER cost estimation tools. For her longstanding contribution to commercial cost prediction tools, the International Society of Parametric Analysts honored her with its 2002 Parametrician of the Year award. McRitchie's most recent endeavors involve building parametric cost models for IT projects and ongoing support. She has a bachelor's degree in mathematics and system science from the University of California Los Angeles, and completed her master's work in mathematics at California State University, Northridge.

Galorath Incorporated
100 N Sepulveda BLVD
STE 1801
El Segundo, CA 90245
Phone: (310) 414-3222 ext. 622
Fax: (310) 414-3220
E-mail: karenm@galorath.com