

Last Updated: 23-March-2004
Joe Befumo,
joe@befumo.com
Shakti Development, Inc.

Quality Doesn't Cost, It *Pays*

Background

When, in 1979, Philip B Crosby wrote *Quality is Free*¹[1], he referred to the notion that the *cost of quality* was inevitably more than offset by decreases in costly rework. Although Crosby referred primarily to manufacturing environments, the philosophy is, if anything, even more applicable to the development of software. In fact, I would contend that it's *better* than free; it *pays!*

Overview

This paper will dispel the common misconception that Q/A is equivalent to a testing program, and will argue that quality is built in through a robust development *methodology*, and not simply 'tested on' at the end. In essence, a well defined, flexible, and dynamic development process represents the cornerstone of the Q/A function.

For all of his valuable insights, Crosby has some limitations when applied to software production in the early twenty-first century. For one thing, it's born of the 1950s and 1960s militaristic management style. Such an approach is unlikely to be well-accepted by workers in any field, and more so among the creative spirits that represent the overage software developer. This brings us to our first area of insight, and potentially the most serious impediment, if not handled properly.

Basic Problem

Paperwork and documentation are anathema to software developers. Their joy is in the sheer exhilaration of creating intricate, dynamic constructs that will bring delight to the lives of their users. Nonetheless, this enthusiasm must be reigned in slightly, but without dousing the spirit. There are four 'kid-glove' approaches to remedying this situation.

¹[1] *Quality is Free*, 1979, New York, Mcgraw-Hill.

Mitigation #1

The first approach is for management to absorb the greatest possible amount of the dreary paperwork, in order that those doing the real work can do so as unencumbered as possible. In my “Seminar for New Managers at Digital Equipment Corporation), taking *many* years ago, one piece of advice stands out among the rest: “Delegate what you’d most like to do yourself, and do yourself, what you’d most like to delegate.” This advice has served me well over the years, and has a direct affect on the success of any Quality Initiative. This is no place for an ivory-tower egotist who considers detailed work beneath him or her!

Mitigation #2

The second approach deals with how we introduce and present that additional overhead which, by its nature, can not be done by any but the technical resources. I find myself uniquely qualified to address these issue, having been on both the methodology-writing end, as well as the position of having to adhere to it as an already stressed developer.

The bottom line is that whatever is required of the developer must be shown to actually free them up for more creative endeavors, *to wit*, that some up front diligence will serve to relieve them of countless frustrating hours of bug fixes, rewrite, and modification—inevitably just when they’re hot to dive into an exciting *new* project.

Mitigation #3

The Third approach is to make the guidance information for employing these newly mandated tasks easy to access. Dusty volumes of methodology content, stacked on a shelf at the end of the office, are rarely embraced. Content must be lively, active, and make it easy for the user to quickly locate the are of interest, as well as easy to use templates and examples that relieve him of all but the most perfunctory participation.

Mitigation #4

Finally, the fourth is to actually work with the project teams, training them in the usage of the tools and Techniques, and, where necessary, rolling up sleeves and picking up on some of the development slack to go through the new steps, which, being unfamiliar, will take longer to do the first few times.

Philosophy

The underlying concept of this process is to find non-compliance items at a particular level, at such time as that con-compliance will be the least expensive to

address. The cost of resolving a discovered defect is exponentially related to the distance between the point at which the anomaly is introduced, and that at which it is found. Stated simply, if you build the wrong thing (requirements defect), and don't discover the fact until acceptance test, you're going to largely have to rebuild the entire system. Conversely, a design problem recognized while still in the design stage usually entails no more than adding, deleting, or modifying a class, changing a connector, etc.--not too painful at all.

The problem is exacerbated by a widespread predilection to do things in the wrong order. The typical scenario would be as follows:

It is decided that an upcoming project will be written using Microsoft's new .Net framework. It's just come out, and nobody on the team has any but the most superficial concept of what it is. Typically, the first thing people do is to take some existing code and try to get it to work in the new environment. After some tweaking, the code works, so in order to mitigate what is perceived as a significant project risk (new technology), the team gets together and starts writing some of the low level components they *know* they're going to need: database interfaces, low level business logic, etc.. While they're at it, they implement some things that might be "nice to have," even though they weren't included in the original specs. In proceeding thus, however, they have totally missed critical nuances in the .Net architecture. They have inadvertently implemented all of their typical page-heavy code in traditional .ASP pages, not availing themselves of the "code-behind" features of .Net.

As the truth is discovered, they have already dedicated significant portions of project time to the early efforts, and can no longer afford to rewrite them without significantly affecting project deadlines. So they plow forward, maybe drawing a few quick white-board lines and bubbles, then pounding out higher level program logic, whose structure, by the way, is now dictated by the interfaces casually provided by the original "experimental" components.

Meanwhile . . . requirements changes, as always, begin to trickle in. If the team were still doing use-case modeling, or even class design, such changes would be easy to incorporate; however, they are now heavily invested in code, and the changes will have ripples throughout the system. Deadlines are drawing near, and the crew pulls late nights to plow through the code and add the needed functionality.

The code is finally "thrown over the wall" to the testers, and (inevitably), bugs are found—some simple, some involving logic that will certainly be a bear to fix. The project is not past deadline, and becoming increasingly over budget. Ten-, twelve-, fourteen-hour days follow one upon each other. Everyone is frazzled, and tempers flared. Stupid mistakes are made, simply because of fatigue.

But this is a team of superstars, and, against all odds, they manage to finally get a finished product released. Yes, it's brittle, over budget, and, yes, inevitable maintenance will be a nightmare—costing many times what it would for a better-designed product—but, by gosh, they did it!

Sound familiar?

One problem here is that, ultimately, you have paid more than the price that would have accrued had you gone through a quality-oriented process, but you have not received the quality you have paid for.

The architectural similarity is having a contract to build a house with a tile roof. Having never worked with tile roofs before, you decide to build a framework and build the roof first . . . just to be sure it doesn't become a bottleneck. . . .

A Better Way

Now, let's look at how this *might* have been approached. First, we agree on a few things:

1. Quality does *not* consist of delivering a Cadillac when a Ford Escort is needed. Nor, does quality refer to building a vehicle that will go one-million miles between tune ups. Quality refers to building *what* the client has ordered, to a predictable level of reliability (the level of reliability needed for controlling a Nuclear Plant is far more stringent than that needed to control a lawn mower's engine speed), within a predicted cost and schedule.
2. We are *not* dealing with a fundamentally novel kind of technology (e.g., biological computers). We are dealing with a new portion of an existing operating system. Yes, we can expect it to have bugs, and will work this into our risk/contingency plan. Nevertheless, we can assume (after visiting some beta .Net websites) that it can be made to work. We needn't start experimenting with its low-level capabilities just yet. (As an aside, I can state that in 20+ years of software project work, I have *never* seen a project fail because of purely technological reasons. Technology can be taken for granted in 90% of the situations).
3. We can not reduce an estimate/schedule by eliminating seemingly redundant aspects of the methodology. The result will take as long as it's going to take anyway, and the end result will be an inferior product. Hey, wait a minute . . . how do we come up with that estimate anyway?

Process and Estimating

Process and estimating are so intimately intertwined as to be inextricable. *Nobody*, no matter how good they are, can look at a six-month project and identify it as such. Some people have tried to address this issue by constructing a mammoth, all-inclusive project-plan template, which, they anticipate, can be applied to every project. The problem with this is that, as has been quantitatively proven, the complexity level of projects as a function of elapsed time is not a linear, but more closely, an exponential one. It's relatively easy to build a shopping-cart site, a word processor, or a new mail client. It's a whole different thing to automate the Denver Airport.

Thus, a flexible, dynamic *Methodology Deployment Environment*, will distinguish between the levels of complexity inherent in various categories of project. Moreover, the tool must allow project planners to add and remove components that may or may not be applicable to a given project.

Hey, wait a minute! Didn't I just say that the project plan cannot be mangled?

Well, the answer is that when project setup of this sort is done as a group, it is easy to identify shortcomings, and come up with a well detailed, but customized plan. In fact, there *are* times when unfortunate shortcuts must be made, and a methodology that does not allow for this is not a good methodology. For example, a startup company may have enough operating capital for six months, and if they don't get a certain set of functionality to the market in that time, key investors will shut them down. Fine! You have to do what you have to do. Nevertheless, a good methodology will guide you through understanding and quantifying the risks involved by shortchanging certain aspects of the recommended process. Hence, in preparing their pro forma for the next several years, planners will need to include additional maintenance and enhancement costs for the less reliable system that results. This, however, is a business *decision*, rather than (as it too often is) an unconsidered consequence.

Process, Estimating, and Metrics

So, what *is* methodology? To answer that in a comprehensive manner is outside the scope of this treatise. Consequently, I will deal with one component of a methodology: The Process Roadmap.

The process roadmap will consist of a hierarchical set of process building blocks, some combination of which can be made to represent a particular project. Hence, you may have project kickoff modules that are suitable for a small utility development project, and others that are aimed toward huge enterprise-wide initiatives. You might have design modules aimed at complex object oriented systems, and small report generation undertakings. What the methodology will do for you, however, is ensure that you put together a full

complement of process segments to represent a well-designed project. (Of course, once again, the planners are always free to circumvent these recommendations if necessity is deemed sufficient.)

What these modules give you is a hierarchical breakdown of activities, tasks, deliverables, and Q/A gates at a very detailed level. All of the better methodology tools include estimating capabilities. This, of course, is not like pressing a button and having a program generate a cost and schedule. However, unlike looking at a six-month project and trying to “guestimate” a figure, it is easy for a group of experienced professionals to come to an agreement on how long it will take to code a single object with n number of operations.

Quality and the Estimate

This brings us full circle, back to that all important quality function.

In preparing a detailed estimate, our methodology will show us where (and how) we will be doing peer reviews, where structured walkthroughs are appropriate, and when a major phase audit is called for. Moreover, it shows us how much time we should allow for the Q/A action, the resulting rework, and the subsequent re-assessment. It tells individual contributors how much time they must set aside for peer reviews. (In other words, it is *everybody's* job to ensure that a quality product goes out the door.)

This is *extremely* important, because one of the most prevalent excuses for not performing these quality activities is: “We just don't have time.” In reality, what is meant is “We just didn't *plan* for it.” Well, guess what? Q/A isn't just something that “nice to have” on a project. It's as integral an aspect of the development process as doing UML Class Models, physical data modeling, or writing code!

Testing

As you might have noticed, I've yet to say anything about *testing!* That's because, in my opinion, testing is the least significant contributor to the ultimate quality of the product, as well as the most expensive one. In order to mitigate this expense somewhat, automated testing tools are a must. Yes, they're generally expensive to buy, but the investment pays for itself many times over in relatively short order. Once again, if the process is followed conscientiously, the bugs found in test are minor and easy to locate and repair. Finally, it should be noted that testing is not something that happens at the end of the process, when everything else is finished. *This* kind of acceptance should be no more than a formality. *All* intermediate deliverables

are subject to quality review or testing, as appropriate. And, even then, few problems should be found, because the system of programmer awareness, and peer review ensure that the most obvious flaws will be located long before formal Q/A actions.

Biographical Sketch

Joe Befumo is a Senior Managing Partner and Chief Technology Officer at Clarity Development, LLC. After working in software development since 1974, he became interested in process improvement issues in 1989, while at Digital Equipment Corporation's Advanced Semiconductor Development Group. As a methodologist and head of AT&T Solutions' Center for Estimating and Metrics he contributed to the development of software processes geared toward the large-scale distributed development projects. He has been Chief Technology Officer of Clarity Development since 1997.
