Optimal Attention to Software Quality:

If You're Guessing,

You're Probably Wasting Money

Bill Tepper

November 14, 2005

Provaré Technology
5174 McGinnis Ferry Road, #259
Alpharetta, GA 30005

E-mail: info@Provare.com Web page: http://www.Provare.com/

Table of Contents

1. Introduction

Software development is one of the most complex and high-risk endeavors that any business can attempt. The rates of change in the marketplace and technology that confronts most software projects is enormous. To overcome these challenges, decision makers must make use of a combination of business, project management, marketing, and technical skills that no other industry in human history has ever required.

Each of these four skill sets has a different focus and each has developed its own unique language to communicate its particular point of view. Finance types speak of revenues, costs, expenses, investments, and profits. Project managers are more interested in schedules and the best use of scarce resources. Those focused on marketing are more interested in market conditions, product appeal, product niches, salability, feature sets, competition, and customer reactions. Engineering managers are quite content to leave these issues to those in the "front offices," instead preferring to focus on the technical challenges of developing a product with a rich, cutting edge feature set and having fun doing it. It is rare to find leaders who speak more than one of these languages fluently. This difficulty in communicating between these fundamental business disciplines pervades technology development, but seems to hit software development particularly hard.

This nexus of communication, while a major source of both critical product defects and product development opportunities, is almost completely neglected by the processes favored by each individual discipline. There is only one branch of software development whose primary focus ***is*** this very nexus – Software Quality Control. Remarkably, what may be the most profitable aspect of software development is often the most neglected.

No one wants to overspend on product development. But the costs of releasing defect-laden software are *significantly* greater than most people assume. Many development managers intuitively "know" this to be true, but relying solely on intuition is a costly mistake – as will be illustrated by several real-world examples in **Section 3**. The good news is that there is a goldmine of data right at our fingertips that will guide us to the most profitable level of attention to software quality. All we need to do is use it.

There are three primary opportunities for quality improvement in a development organization:

- ✓ **Defect Prevention** – preventing defects from reaching system test.
- ✓ **Defect Discovery** – Finding and documenting defects before they are released to customers.
- ✓ **Defect Correction** – Correcting defects that are discovered and reported.

There is an optimal attention to – and expenditure on – each of these areas. Since we are discussing product development in the context of a business, when we say optimal, we mean most profitable, of course. Let's examine the income and expense contributors to this optimal point…

2. Optimal Quality is Optimal Profit

Intuitively, it is not hard to imagine a case where a development organization is so indifferent to quality that its product is entirely unusable. On the other end of the spectrum, we can imagine an organization that consists only of test engineers and process supervisors, but with no product actually being produced. Given the conceptual existence of these extremes and given the fact that technology products **do** get produced, sold, and used in the real world, we may conclude that somewhere between these two extremes must lay a realm where profitable development takes place.

2.1. Why Aren't We All Optimal?

Given that level 5 of the SEI's Capability Maturity Model is entitled "Optimizing," there is probably a broad consensus that software development can be optimized in some sense. This topic has been studied, discussed, and preached in a variety of forums and from a variety of angles for many long years. And yet the SEI still estimates that the bulk of software shops are CMMI level 1 or 2. None of the organizations with which the author has any level of experience ever actually behaved in a manner suggesting any awareness of the benefits of self-examination and optimization. Nor have any been at a CMM or CMMI level greater than 2.

So we are left with a puzzle of sorts. Do the research and writings lack credibility? Does the typical software engineering or product development manager simply not know about these options? Is it merely a perception problem? It may be a combination of many factors, but consider the following possibility: the unintentional academic emphasis on process rather than on common sense behaviors has caused many, if not most, development managers to throw their hands up in resigned despair before the battle has even begun. They reason (or simply feel) that they do not have the bandwidth or the energy required to cause their teams to adhere to the onerous processes that they believe are required for achieving or even approaching optimal quality levels with their scarce resources, and they never give the matter another thought. In a very real sense, then, the perfect has unwittingly become an enemy of the good.

It is the assertion of this author that any manager of any technology product development effort can, with a few straightforward steps, make significant progress toward optimizing his or her emphasis on product quality, thereby substantially improving the profitability of his/her product.

2.2. Defining Quality

Although this is admittedly a relatively narrow definition, for the purposes of this analysis, we shall define quality as:

> Quality – a measure of the extent to which the product meets its defining requirements without defects.

This definition allows us to propose the following hypothetical quality boundaries:

> Perfect - The product meets or exceeds all defining requirements and has zero defects. Quality measure is 100%
>
> Perfectly Useless - There is no correlation at all between the product's functionality (if any) and its defining requirements. Quality measure is 0%.

2.3. The Costs of High Quality

With these definitions in mind, consider the chart of quality level achieved versus amount of effort (i.e., money) spent on quality in **Figure 1**.

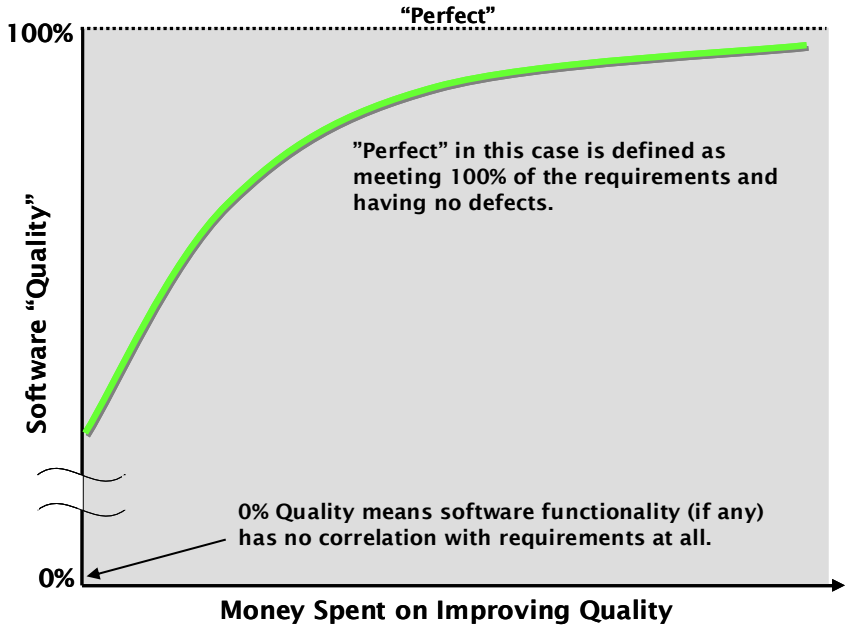

Figure 1. Improvement in quality with increasing expenditures on quality

Figure 1 Illustrates what our experience tells us: that it is relatively inexpensive to make significant gains in quality when first addressing the problem. But as the quality level approaches perfection, remaining defects are more elusive and more costly to find – so we can never quite achieve perfection, regardless of what we spend trying. Since we are looking for the

costs of high quality, let's swap the axes on **Figure 1**. This gives us **Figure 2**, illustrating the costs of high quality.

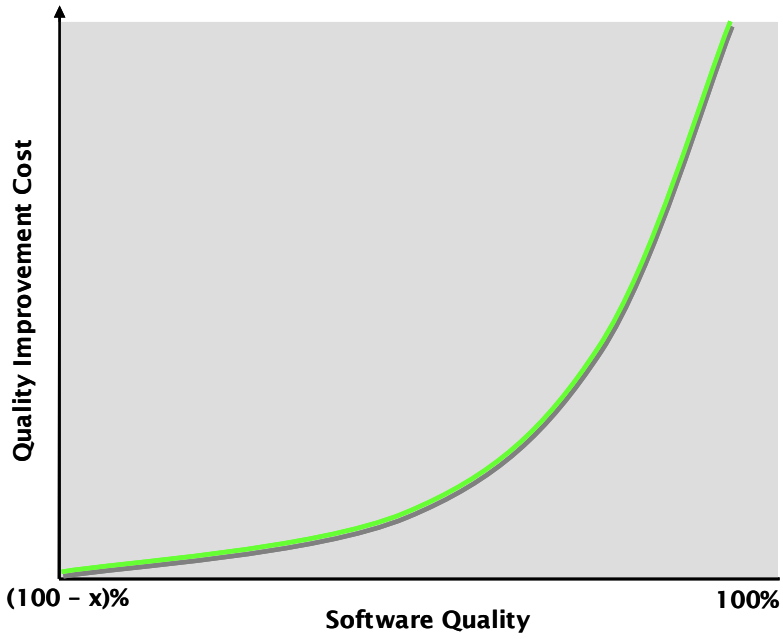

Figure 2. Cost of Quality Improvement vs. Quality

2.4. Costs of Poor Quality

Some examples of the costs of poor quality in a released product include:

- ✓ Increased field and call center support costs.
- ✓ Increased cost of producing and distributing larger numbers of bug fix releases (potentially creating significant waste of previously manufactured product).
- ✓ Increased certification and approval costs in regulated industries.
- ✓ Increased liability exposure and increased liability insurance costs (e.g., in industries such as medical devices, financial transactions, etc.)

But defects are not only costly in released software. Pre-release defects can add costs such as:

- ✓ Increased test time per cycle and increased number of test cycles before desired quality level is achieved (added costs of testing defective software).
- ✓ Increased time to market because of additional testing and debugging resulting in lost revenue.
- ✓ Increased cost of correcting defects.

- ✓ Lost opportunity costs resulting from tying up development and test resources on maintenance instead of applying them to new features or new products.

Any company can (and all companies should) know exactly how much they are spending on support for any given product. Now consider the relationship of support costs to the quality of the product as proposed in **Figure 3**.

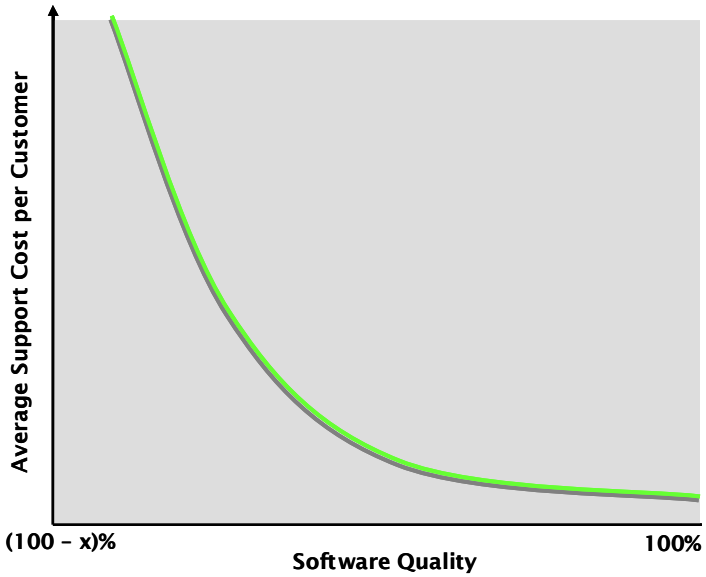

Figure 3. Support Costs per Unit vs. Product Quality

Again, this curve is quite intuitive. As quality of the product increases toward perfection, the costs of post-sales support decrease asymptotically toward some theoretical minimum.[1]

2.5. Total Costs vs. Quality

Now that we have the curves in **Figures 2** and **3** plotted against the same independent variable, we can include both curves on the same chart, and have done so in **Figure 4**. Suddenly these curves look very familiar. In fact, they are very like the supply and demand curves that we all remember from Economics 101. And it starts to become obvious that there will be a point at which costs are minimized.

[1] The theoretical minimum support costs are those that would be necessary to support a perfect product for imperfect customers. So support costs can never reach zero.

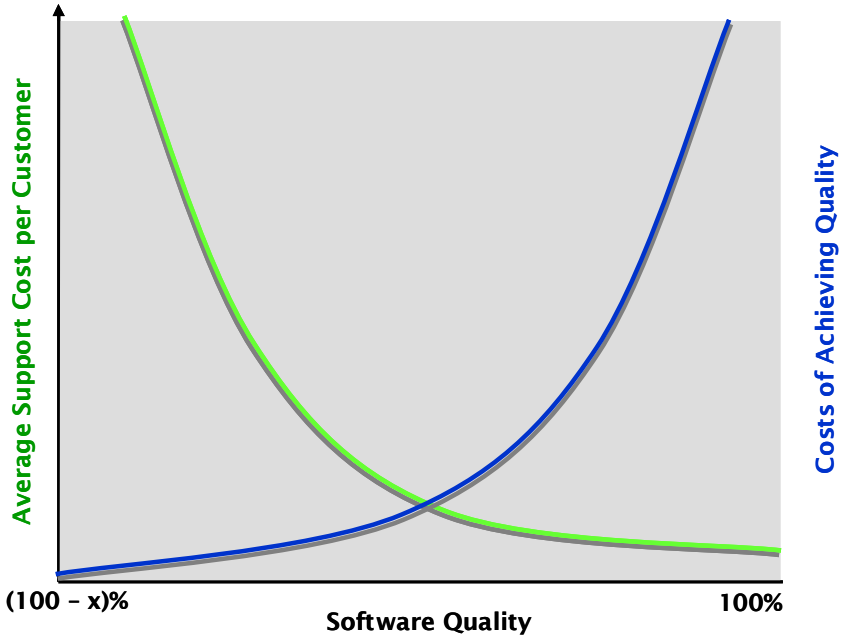

Figure 4. Costs Affected by Product Quality

Since costs of achieving quality and costs of support are both real costs that are additive, we can add them to produce the curve shown in **Figure 5**. This curve makes it quite clear a minimum cost point really does exist.

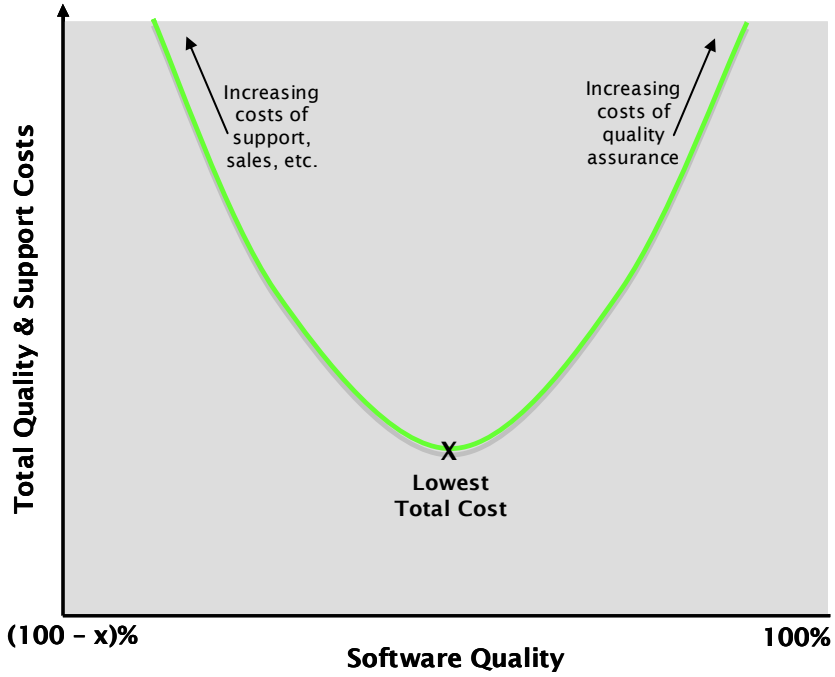

Figure 5. Total Costs versus Quality

2.6. Impact of Quality on Revenue[2]

Some examples of the impact of poor quality on revenue include:

- ✓ Lost sales and increased costs of sales
  - Slower customer rollouts & slower purchase schedules
  - Decreased follow-on business and/or lost customers
  - Damaged reputation preventing some new sales
  - Sales staff spends additional time with unhappy customers that cannot be spent on new sales.
- ✓ Increased customer cost of ownership, resulting in a decrease in what customers are willing or able to pay for the product.

These reductions in revenue resulting from poor quality are devilishly difficult to measure directly, but we can get a very good sense of them by looking at support costs. It is probably a very safe bet that as support costs due to poor quality increase, revenue will decrease accordingly.

According to Technical Assistance Research Programs (TARP) [Goodman, 1999], for every customer who takes the time to complain about a bad experience with a consumer product or service, somewhere between 1 and 19 more customers felt the same way but did **not** complain. Each of these 2 to 20 customers represented by that single support call will tell from 5 to 16 other people about their negative experience. This means that for every complaint you hear, somewhere between 10 and 320 people have a negative impression of your product or service. Of those who **did not** complain, somewhere between 63% and 91% will **never** do business with your company again.

For a business-to-business product or service, the complaint rates are higher – 75% of unhappy customers will complain. On the downside, how many of our potential new clients will they tell? For big-ticket items where proposals are required and references are thoroughly checked, it is a safe bet that they tell ***all of them!***

It is perfectly reasonable to assume that the negative impact on revenue resulting from low quality is proportional to the corresponding increase in support costs. So *for a given fixed release date* we can illustrate the impact of quality on potential revenue as in **Figure 6** (if increasing the quality pushes out the release date, then the curve would peak at a quality level lower than 100%).

[2] This section does not discuss the impact of delay of product shipment on revenue that might result from a long test-debug cycle. For interested readers, this is treated briefly in the Appendix.

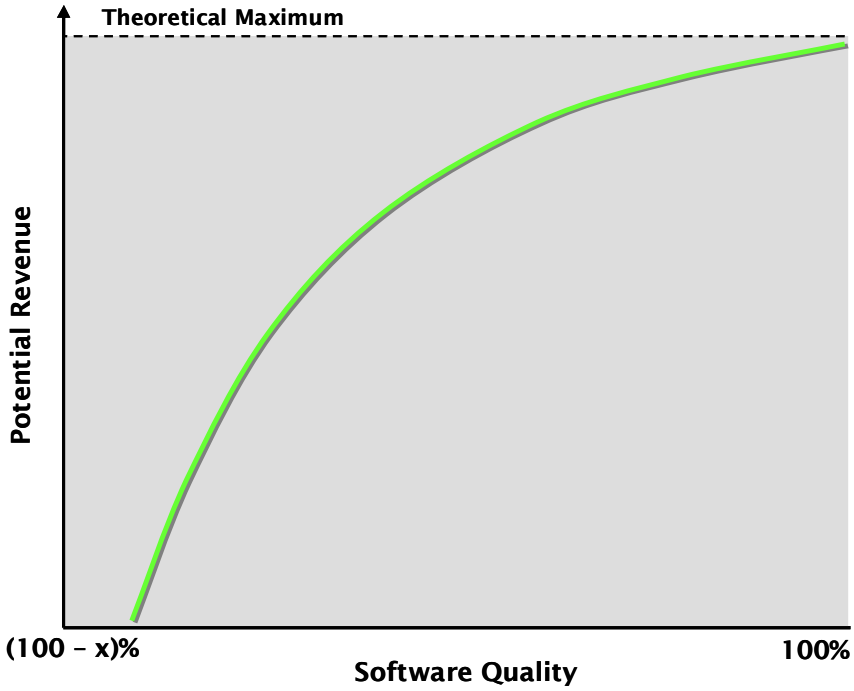

Figure 6. Total Potential Revenue vs. Quality

We can now subtract our expenses (**Figure 5**) from our revenue (**Figure 6**) and arrive at a curve relating potential profit to software quality as shown in **Figure 7**. This curve clearly illustrates how an optimal quality level can maximize our profit. So now we know not only *that* this point exists, but *why* it exists.

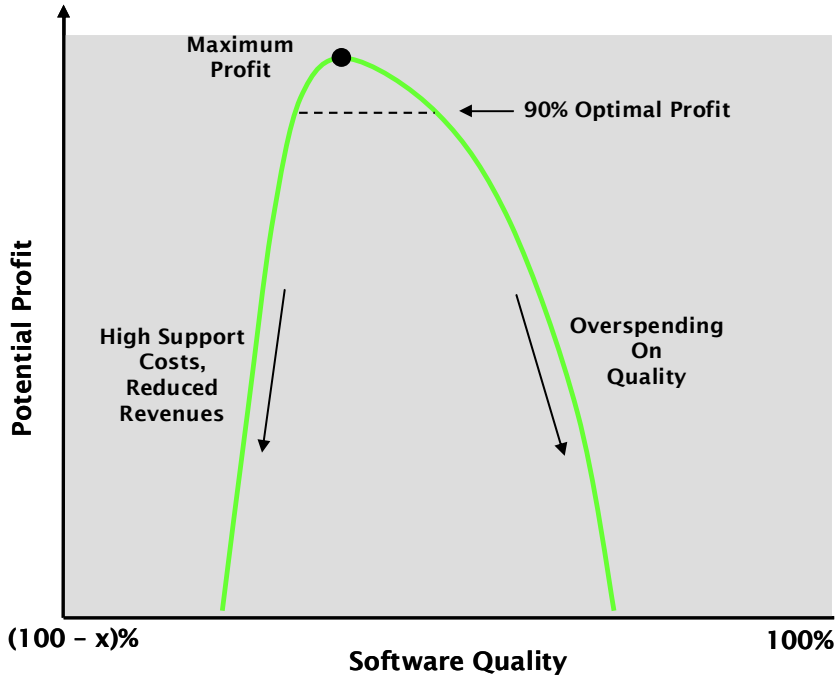

Figure 7. Potential Profit vs. Quality

If we pay too little attention to quality, our revenue will be reduced and, even if we might otherwise make a profit, it will be eaten up by support costs. If we spend too much on quality, the marginal increases in revenue will never cover the additional costs.

We will probably never precisely optimize our attention to quality, but most of us would be thrilled to get 90% of the way there. As **Figure 7** illustrates, we can do so with considerable latitude.

3. The Costs of Failing to Optimize

In the preceding section, we briefly addressed the very real costs of failing to address quality optimization. The best way to demonstrate the error of relying on intuition or guesswork in defect reduction activities is to demonstrate with real-life examples of how wrong a project can go.

3.1. Failing at Defect Prevention

This is probably the most often overlooked area of potential quality improvement. Even the best organizations fail in some way here. The unfortunate thing is that the many of the kinds of bugs that escape this step can be the most devastating to a project.

In one example organization, the developers were quite competent. The product was a communications product requiring that many devices be distributed to remote locations. The product used cutting edge technology and was extremely complex. Designs were often written and reviewed, but code reviews were rare. And as is the case with many development organizations, there was no definition of a "unit test," so if any unit testing was done, it was entirely at the individual developer's discretion.

So defects slipped past development – and some of them slipped past system test into the hands of several customers. And time went by. Eventually, after enough customers called angrily saying that their devices had either mysteriously rebooted or had locked up and stopped serving their customers, the troops were rallied. This bug had to be found and eliminated!

At first, most of the engineering and field support departments were deployed against this bug. But the bug simply evaded everyone. The thing was, it was the kind of bug that does not cause any ill effects at all in the domain of its own code. It was the type of bug that we all know and loathe – the dreaded memory time bomb. The process in which the bug resided ran beautifully well until some rare conditions arose that caused it to write to memory other than its own. This failure had no immediate consequences, but once it happened, doom was inevitable.

So two key developers were told to stay on the trail of this bug until it was found. Code reviews were organized. Code was written to try to detect the corruption as early as possible. Still more code was written to correct the

error when it *did* occur. Many a hair turned gray or was pulled out. But 12 weeks and several hundred man-hours later, although the code reviews had unearthed several areas for significant improvement, the actual smoking gun was never found.

The truth is that it is impossible to know whether or not this bug would have been prevented if better unit testing and/or code reviews had been done. But we certainly know what *not* finding it ultimately cost. Not only did the product suffer a damaged reputation with many customers (who talk among themselves frequently) but many hours of developer time were spent and the bug was never found after the fact.

3.2. Failing at Defect Detection

Another case involves an innovative and complex new communications product, finally getting a handful of customers after a considerable period of development. Prior to release and the first few customer installations, there had been no testing program at all, so all defects that got past the developers were found in the field. Field engineers came from R&D staff, ostensibly because of the complexity of the product.

After only 4 customer installations, problem reports from the field were consuming 100% of the time of the R&D staff (about 14 engineers). Multiple new bug fix builds were being released to the field per day, likely with new bugs being introduced at nearly their rate of correction – there was no way to know. No new features were being developed and no additional customers could be supported at this staffing level. R&D engineering had completely lost credibility with sales, program management, and customers. The R&D staff was exhausted from working 12 to 15 hour days, often 7 days per week.

On the advice of a consultant hired by the Division President, the R&D team added one test engineer, two field support engineers, and one change management (CM) specialist. The CM specialist began by putting a defect tracking tool in place and the test engineer began filling its database. The R&D manager decided to halt all new development and spent 8 weeks doing a release containing nothing except bug fixes. For this release, the team literally fixed every known bug, no matter how small. Except in the rarest of circumstances involving bugs that were otherwise not reproducible, only the field engineers were called upon to travel to customer sites.

After these changes and the bug-fix release, as if by magic, the chaos subsided and, over time, credibility was restored.

This team had never intentionally released a defect to the field. But they had no independent assessment of the quality of their product other than their customers. Once they knew about their defects and made the decision to correct them, sanity was restored.

3.3. Failing at Defect Correction

This entire account was paraphrased from Kaner, et. al. [2002].

At one company, the management was very educated and aware of quality issues. They had taken most of the recommended actions to prevent defects and to find them. But because of the pressures of management and the marketplace, they always deferred the minor defects – i.e., those that did not really reduce the functionality of the product, but which caused it to do unexpected things or were just plain annoying.

At one point in time, once the product had been released for quite a while, the change control board decided to review the technical support call log. To their great surprise, an analysis of these calls indicated that over half of the technical support call hours had been spent on those many known, trivial bugs that they had consciously decided not to fix. They postulated that they could cut their support costs in half by fixing these bugs.

To test this hypothesis, they halted all new feature development and spent the necessary time to fix all of the trivial bugs for which calls had been received. They were surprised at how little time this actually took.

Once this bug fix version of the software was released, true to their prediction, customer support costs fell by half.

3.4. Failing All of the Above

Although it seems impossible to believe, there are companies who introduce very innovative products that fill a critical market niche, but who fail to truly address quality at any level. One such very small startup was in its fourth year of business and had even reached cash flow positive status. And yet, there was no independent test department, no review of tech support calls, no post mortems, etc.

It was not that the CEO was indifferent to quality. In fact, when the suggestion was made to him at one point that he needed to establish a testing program, his response was "what do you mean there are bugs in our software?!!!" He was serious. He thought that every effort was already being made to assure not just quality software, but *perfect* software. And yet he had never heard of independent testing, even from his technical managers, who should have known better.

At that time, about 80% of the company's staff was technical (i.e., engineering, IT, and tech support). Somewhere over 80% of that technical staff's time was spent handling emergency tech support calls, up to and including the time of the CTO and VP-level managers. Many of these calls were coming though the CEO's office. Since the technical staff were already more expensive per person than average, this meant that well over 2/3 of the company's entire salary budget was being spent on customer support!! As a

result, this technical staff worked long hours, suffered terribly from burnout, and the turnover rate was more than 50% per year even though the company was paying well above market average salaries.

Eventually this situation was turned around. The company still exists and is profitable and growing. But the transition was painful, involved a lot more turnover, and a slow process of education of the executive management. It could just as easily have turned out very badly for everyone involved – and does for countless such startups every year.

3.5. Incredible, but True

Even as I wrote the preceding section, I found the stories incredible. Had I not personally witnessed them or heard them from very reliable sources, I might not believe them. Even with the vast amount of information on how to prevent such disasters, these stories are the rule rather than the exception – at least in the commercial world.

Each of the product managers had believed they knew when "enough" defect correction had been completed. The truth, however, is that they were shooting from the hip rather than taking the time to look at their situations objectively. In each case, significant resources were expended after the fact to correct issues that should have been addressed prior to release.

I really wanted to include a case study that demonstrated an overemphasis on quality, but I was unable to find a good clear-cut example. Of course, we have all heard of huge programs (my experience is with DoD) that are eventually cancelled after years of missed deadlines and/or missed quality goals. But there seems to be nearly universal agreement in the literature (and I agree, by the way) that these cases were not so much due to overemphasis on quality as they were on poor choice of development model, poor execution, poor definition of the end goal, etc. My few experiences with cancelled products in the commercial world have been clear failures of marketing, not of development (i.e., there was never sufficient market for the product in the first place and this was known only after it was introduced).

Maybe this point is instructive. In his book entitled "Beyond Fear," [Schneier, 2003] Bruce Schneier reminds us of the disparity between our gut feeling for the frequency of scary events and their actual odds of occurring. He points out that 40,000 Americans die in auto accidents every year and that to produce this many deaths from air travel, the equivalent of one 727 would have to crash every 36 hours. Yet Americans universally take driving for granted while most of us are at least somewhat unnerved by air travel and many refuse to fly for any reason. We should consider the very real possibility that our fears of losing revenue by delaying the release of a product and of "wasting" precious resources on improving quality are an example of this imperfect feel for real world statistics. We have very little visibility into the actual monetary effects of market timing, but we know that missing it badly

would be catastrophic, so our fears may drive us to vastly overestimate its importance relative to quality. The importance of avoiding this trap (and many others) is discussed at length in Hammond [2002]. The prescription for avoiding it is to recognize that you may fall prone to it and to gather as much objective data as possible.

Given the overwhelming evidence in favor of a small amount of proactive attention costing considerably less time, money, and resources than was paid as a consequence of doing nothing, such situations are simply inexcusable. Yet many organizations repeat mistakes like these on a daily basis.

4. Opportunities for Quality Improvement

In **Section 1**, we introduced our 3 primary areas of opportunity for quality improvement: Prevention, Detection, and Correction. Now let's consider these 3 areas of opportunity separately and look at some factors that affect how successful we can be at each of them.

4.1. Opportunities to Prevent Defects

Table 1 lists 21 rows of factors that have a direct impact on the creation/prevention of defects by a development staff. This table also includes 7 columns that identify key influences on these 21 factors. The items in these 7 columns may be thought of as "control points" through which the needed adjustment of the 21 defect prevention factors may be accomplished. It would be tidy indeed if the 21 factors could be neatly divided into 7 groups, each identified by one of these key control points, but the truth is that most of them are affected by several control points and all but 3 are impacted by more than one.

The key control points are briefly defined as follows:

- ✓ Personnel – the engineering staff itself, including its makeup, training, education, experience, skill, attitude, etc.
- ✓ Management – the technical management of the company, including anyone with management responsibilities, and what this management team emphasizes, requires, and does.
- ✓ Process – Written or unwritten policies and procedures describing what is to be done and how.
- ✓ Culture – The less tangible aspects of the way the members of the engineering staff treat each other, the way they view and interact with customers, etc.
- ✓ Requirements – The authoritative source of information about what the product is supposed to do and how it is supposed to do it.
- ✓ Tools – Any and all hardware, software, etc. that the engineering staff can use to improve their ability to complete their assigned tasks.

- ✓ Product - the product itself, including its complexity, its interoperability requirements, its user interface, etc.

Table 1. Factors Affecting Defect Prevention

Defect Creation Factor	Personnel	Management	Process	Culture	Requirements	Tools	Product
Experience and/or education of developers	X						
Experience and/or training of developers with tool set, OS, compiler, or third party software being used.	X					X	
Realism of development schedules		X	X				
Extent of developer unit and integration testing	X	X	X	X		X	
Clarity and realism of requirements					X		
Stability of requirements		X	X		X		
Flexibility of process and schedule to handle the unanticipated		X	X				
Presence and quality of high-level specifications and/or designs		X	X				
Development staff acceptance of responsibility for quality, regardless of presence or competence of separate testing staff.	X	X		X			
Usable and reliable build tools and process			X			X	
Presence and usability of source control tools and process.			X			X	
Level of resistance of development team to "gold plating."	X	X		X	X		
Amount of customer or key stakeholder feedback during requirements, design, and early prototyping phases.		X	X		X		
Caution exercised in modifying poorly understood or poorly documented code.	X	X	X				
Code base size							X
Availability of helpful development tools (CASE tools, simulators, emulators, debuggers, etc.)						X	
Management emphasis on quality (with accompanying rewards).		X					
Familiarity of UI designers with domain in which product will be used.	X	X					
Fluency of UI designers in language in which the UI is being written.	X	X					
Willingness of developers to share information, help each other, take responsibility, etc.	X			X			
Resiliency of software architecture to evolution and change.					X		X

Notice something very interesting from the table. If we count the number of defect prevention factors over which each control point exerts influence, we get the relative influences depicted in **Figure 8.**

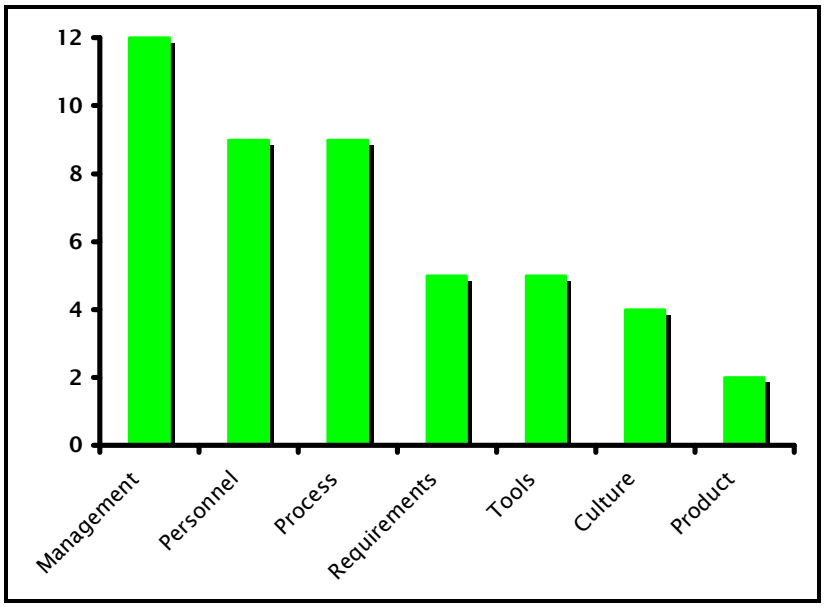

Figure 8. Relative Spans of Influence Over Defect Prevention

Right away, some very interesting conclusions leap out at us. First, as much as we might like to lay the blame for defect creation on the complexity or uniqueness or past mistakes made with the product itself, the truth is that the product itself plays a very minor role in affecting the number of new defects that will be created as it is expanded, enhanced, or debugged.

Second, notice that management has the highest potential impact. This is great news for any manager who identifies with the feeling of despair that we mentioned in the **Introduction**. Managers have more influence than any other factor. And since managers generally have the ability to select their personnel, **Figure 8** vastly understates management influence. The moral of the story is: if you as a manager want a more profitable quality, the power to achieve it is in your hands.

Only a little less important than management in effecting the creation of defects are personnel and processes. It should be obvious that it is critical to get your people on board with your quality efforts. We'll discuss how to do that more in **Section 5**.

And, alas, process rears its ugly head again. But before we slip back into despair, let's step back and take an objective look at what this table is really telling us. Although there are many opinions in this area, there are some areas of very strong agreement that have emerged over the years:

1. A good process is easy to understand and as easy as possible to implement.

2. A good process is flexible enough to handle the unexpected and to allow smart people to use their discretion, but defined enough to keep such discretion from devolving into laziness.
3. The emphasis on quality is completely integrated into a good process. It is not a separate consideration.
4. A good process provides abundant feedback to team members, to management and, as far as is reasonable, to customers.
5. A good process is actually followed – not just put in place as a smoke screen or a "feel good" measure.
6. There are crystal clear reasons behind each and every aspect of a good process and all of these reasons are communicated to team members early and often. Any action prescribed by the process without a compelling reason behind it should be eliminated.

Beyond these few guidelines, the best process for your organization is the one that works for your organization. So don't let yourself be weighed down by the need for a process. Put something into place that works for you and change it when the need for change becomes obvious.

4.2. Opportunities to Discover Defects

Table 2 lists some of the factors affecting defect discovery and again indicates key points of control for each factor. In this case, there are 23 detailed factors.

Again analyzing the levels of influence of the seven primary points of control, we arrive at the breakdown shown in **Figure 9**.

Immediately, we notice that management and personnel are even more important to defect detection while the product itself is less important. Given the examples of quality failures from **Section 3**, this should come as no surprise. It takes a *deliberate* management decision to put a test team into place. The manager and members of this team must have a mindset that is quite unique among software professionals. So it seems clear that if we want to find our defects before our customers find them, we need people whose mission ***is*** to find defects. Moreover, they must be the right people and management must support them.

Once a superior test team and its management are in place, then we can move on to worries about process and culture. Of course, the process guidelines listed in the preceding section also apply here. But note that team culture is of much greater importance in defect detection than it was in defect prevention. Here's why…

Table 2. Factors Affecting Defect Detection

Defect Detection Factor	Personnel	Management	Process	Culture	Requirements	Tools	Product
Independence of test staff		X	X				
Experience of test staff with good testing practices.	X	X					
Experience of test staff with software in the domain of that under test.	X	X					
Emphasis on functional test coverage.	X	X	X		X		
Effectiveness of test planning.	X	X	X				
Appropriateness of test staff assignments vis a vis staff strengths.	X	X		X			
Availability of helpful test planning and management tools.						X	
Availability of helpful defect reporting tools.						X	
Level of management support to the mission of the independent test team (with appropriate reward system).		X					
Availability of specialized tools for execution of tests (highly dependent on product under test, but may include anything from frequency analyzers to simulators to network traffic generators).						X	
Experience and training of test staff with various tools.	X	X				X	
Understanding by test staff that their job is not to prove that the software works, but that it **doesn't** work (i.e., to break it before the customer does).	X	X		X			
Ability and willingness of test staff to test beyond written requirements.	X			X			
Acceptance by test team of responsibility to fully assess (but not to assure) the quality of the product.	X			X			
Test team access to complete and stable requirements.		X	X		X		
Test team involvement in requirements analysis and review.	X	X	X	X	X		
Test team involvement in scheduling.	X	X	X	X			
Realism of time and resources allocated to testing		X	X				
Repeatability of tests performed.	X						
Usability of the product under test (the harder it is to use, the harder it will be to test).					X		X
Existence of alpha and beta test programs.		X	X				
Culture of teamwork among test engineers.	X	X		X			
Culture of teamwork between test and development staffs.	X	X		X			

If a test team is doing its job well, they are producing a lot of what developers would consider bad news. Nobody likes to have his or her mistakes made public, but that is *exactly* the mission of a test team. So to minimize the potential emotional impact of this situation, it is important that both developers and testers have a positive, professional, team-oriented, thick-skinned attitude. Management must not only model this behavior, but must insist on it in their teams and must nip any adversarial attitudes in the bud. Also remember that if developers' mistakes are not found by the test team, then they will be found and made ***much*** more public by our customers!

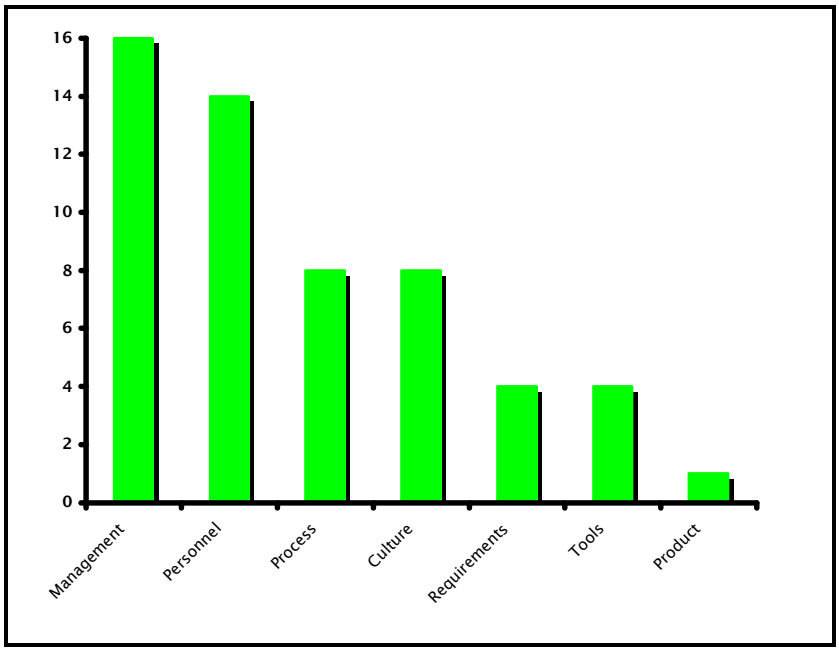

Figure 9. Relative Spans of Influence Over Defect Detection

4.3. Opportunities to Correct Defects

As with prevention and detection, there are several factors that contribute to how quickly and inexpensively reported defects can be found and corrected. Some of the more important ones are listed in **Table 3**.

There are 12 factors listed in **Table 3**, with the key control points breaking down as shown in **Figure 10**.

Once again, management and personnel far outweigh the other control points on the list, but culture has moved even higher in the list. For debugging efforts, the cultural influence is primarily a question of how closely the development and test teams are willing and allowed to work together. In a close-knit culture with a high level of cooperation, test engineers can make significant contributions to the debugging effort. If this teamwork is not present, debugging efforts are much less efficient.

Table 3. Factors Affecting Defect Correction

Defect Correction Factor	Personnel	Management	Process	Culture	Requirements	Tools	Product
Debugging skills of development staff	X	X					
Repeatability of reported defects	X		X				
Number of coexistent defects per function point (large numbers of defects will exhibit correlated symptoms and make debugging diffucult).							X
Availability of adequate debugging tools.						X	
Readability and maintainability of software base.	X	X	X	X	X	X	X
Development staff turnover.	X	X		X			
Ability of test engineers to assist in debugging efforts (if asked).	X	X		X			
Availability of large or unique hardware configurations for reproducing rare defects.						X	
Management emphasis on elimination of defects as a measure of product quality accompanied by appropriate rewards.		X			X		
Cultural recognition of maintenance activities as key to the success of the product (i.e., if the "good" developers always get assigned to the new products or features, then maintenance must not be as important).	X	X		X			
Extent of customer support feedback into development priorities.		X	X		X		
Culture of teamwork between test and development staffs.	X	X		X			

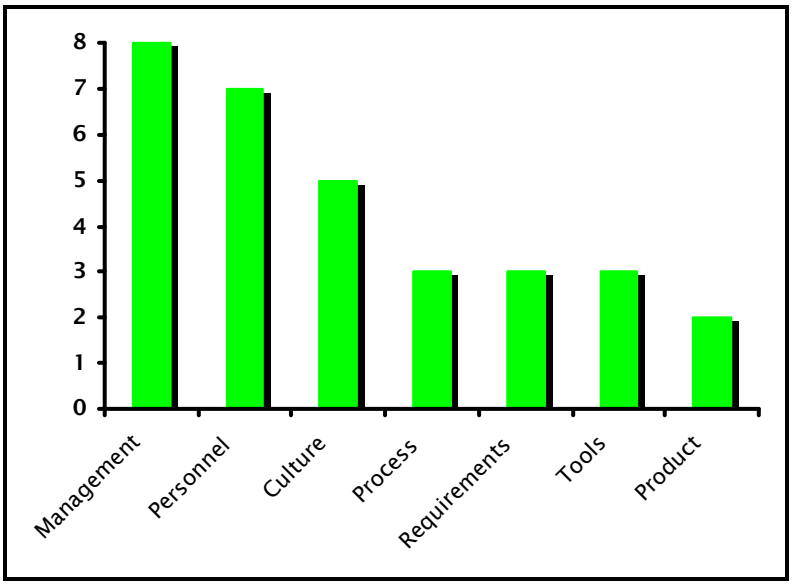

Figure 10. Relative Spans of Influence Over Defect Correction

5. Getting Objective Data

At this point, we have established that:

1. An optimal level of quality investment exists, and we risk losing significant profit potential if we miss it.
2. Despite abundant proof of the importance of quality, most development shops seriously neglect some aspect of it.
3. The neglect of quality will negatively impact our products, customers, businesses, and lives.
4. There are straightforward steps that can be taken to prevent defects, to ensure the discovery of defects, and to correct defects, each connected to one or more identifiable areas of responsibility.

Yet the question remains: We know about all of these things, and we are doing many of them, but how do we know which ones are the most cost effective? There is no perfect answer to this question, but there are ways to get feedback that will move a project in the right direction.

5.1. Feedback on Defect Prevention

The primary metric for measuring the effectiveness of prevention efforts is defects per unit of software functionality (defects in this case being limited to those reported after a build has gone to system test).

Much has been written over the years concerning how to fairly and accurately measure software functionality. Function point analysis seems to have the most venerable pedigree and the most flexibility [Garmus & Herron, 2000]. The problem with function point analysis is that significant training, expertise, and experience is required before a software professional can really do it accurately. That may be the case, but in some cases where software products have very large distributions, it is probably well worth the investment, especially given the potential variety of uses to which the technique may be put. Even if we cannot afford the investment in rigorous function point analysis, we can probably get closer than a wild guess by calibrating simpler counting techniques to spot function point analyses.

The defect-per-functionality metric should be done on a per “module” basis (a “module” being a block of code encompassing a single relatively small and independent set of features) and on a per developer basis. If we measure this on a per developer basis, then we really should also measure developer productivity as well (units of functionality added per month or similar measure). If we don’t, our metric will improve our defect rate, but will also slow production.

A lot of very experienced people in the software development and quality fields argue against using metrics that gauge individual performance. These

experts' experiences are perfectly believable but their prescriptions are dead wrong. First, to say that a software team should not use a metric because its members will find ways to cheat and distort it seems to be tantamount to making excuses for lying, cheating, irresponsibility, and immaturity. These issues should be addressed separately. Honesty and integrity are requirements of any profession and software engineering should not be excepted. Management should expect and accept no less, regardless of the process chosen. Secondly, arguing against using these metrics because they are dangerous seems very similar to saying "you can cut yourself with a knife, so never use knives."

Michael LeBoeuf, in his classic book "How to Win Customers and Keep Them for Life," [LeBoeuf, 1987] points out that there are 3 ways for employers to get what they want from their employees:

- ✓ Tell them what you want
- ✓ Show them what you want
- ✓ Measure and reward what you want

It is plain at a glance that these are listed in order of increasing effectiveness. If we tell our developers that we want productivity and quality and then demonstrate to them that we have absolutely no intention of objectively measuring either, they'll get the *real* message loud and clear: quality and productivity are not really important. Moreover, we will have demonstrated ***our*** duplicity, which our team members can smell from miles away. Michael LeBoeuf goes on to say, "if someone tells you that what he does can't be measured, you can safely bet that he isn't doing much."

That being said, we should keep in mind the cautions that have been broadcast by others concerning using metrics, especially when they are tied to personnel. That which we measure and reward, we are guaranteed to get more of and that which we measure and punish, we are guaranteed to get less of. So before we start using metrics like these, we need to think carefully about what warped incentives we might be inadvertently injecting. Only if we are convinced that they are minimal and/or manageable should we implement the metrics.

Just by making these two measurements, we should see an improvement in our quality and our productivity because of the psychological impact. But we also can use the data in a couple of important ways.

The defect-per-unit-functionality metric is a great first-order way to discern how each of our developers is performing where quality is concerned. We should avoid any direct numerical tie to raises or bonuses, but we should take this data into account and our developers should ***know*** that we consider it. We may discover that a developer is a terrific debugger but is a relatively poor unit tester or that another is great at UI design but poor at data structures.

Such serendipitous results are very extremely valuable. We can only improve when we are aware of the need for improvement!

The second and much more important way to use these metrics is to use the per module defect data to identify modules that are good targets for refactoring or rewriting. If one module or feature represents 8% of the code and accounts for 28% of the defects, then we are well advised to review the design of that module.

5.2. Defect Detection Metrics

5.2.1. Defect Arrival Rate

Basic defect arrival rate data can be pulled directly from any good defect tracking tool. Development and test managers should at least look at this metric for potentially useful information. In some cases, the "noise" may overwhelm any trend present in the defect arrival rate and the metric may be useless - but it never hurts to look.

5.2.1.1. The Theory Behind Defect Arrival Rate Analysis

Defect arrival rate (AKA latent defect) analysis is simply an attempt to estimate the number of remaining defects in a product based on the rate at which defects have been detected during system testing. This technique takes advantage of the diminishing rate of returns for testing that we saw in **Figure 1**.

To perform a defect arrival rate analysis, we begin by graphing the defects found per unit time vs. time as illustrated in **Figure 11**.[3] The discrete points in **Figure 11** clearly indicate a trend that looks a lot like the trend in **Figure 1**. The analysis involves first spotting such a trend and then fitting a curve (such as a Weibell, Rayleigh, or decaying exponential) to the data using a method such as a minimum variance least squares curve fit.

If a trend is apparent in the data, the question of what curve to use to fit the data is academic and pointless. As discussed in Hoffman [2000] the data under analysis are not going to adhere to the assumptions that the pure math demands for *any* curve.

[3] The data in Figure 11 are for an unusually long project and are much better behaved than in most real-world projects.

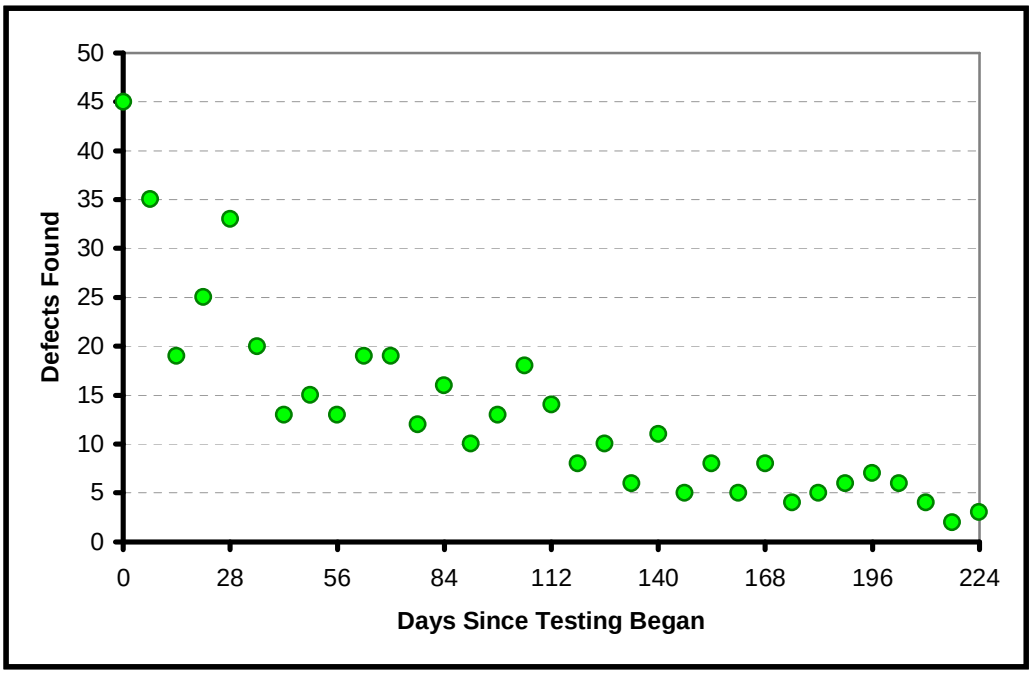

Figure 11. Example of Defect Arrival Rate Data

But this does not matter nearly as much as the fact that the math *will* allow you to find a way to use the data to estimate the remaining defects at any point in time.[4] So we can fit the data with any function *f*(t) that

- ✓ Fits the data well.
- ✓ Decays asymptotically to zero as test time approaches infinity.
- ✓ For which $\int_{T}^{\infty} f(t)dt$ is a finite, positive number.

In fact, there is no harm in trying several curves before choosing one.

Once a function is chosen and is fit to the existing defect arrival rate data, an estimate of the remaining defects is obtained by integrating the curve from the last data point to infinity as illustrated in **Figure 12**.

[4] Just bear in mind that the actual uncertainties are a little higher than the mathematical results will suggest because your model is not perfect.

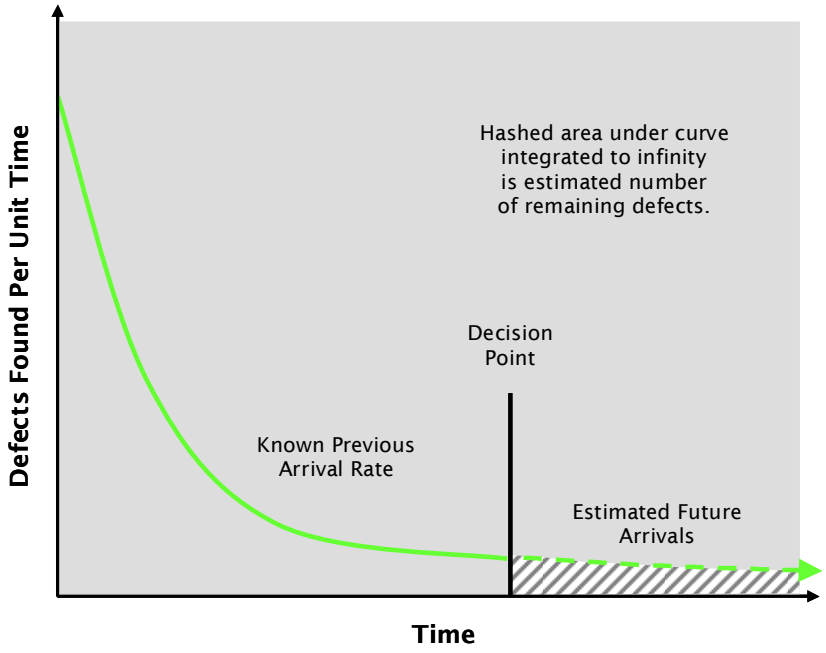

Figure 12. Defect Arrival Rate Analysis

5.2.1.2. A Short Test Project Example

Let's look at the application of this analysis technique to a very short (totally fictitious) test effort. Let's say that, so far, our test team has been testing for one full week. In one week of testing, the test team has found 39 defects. The defects have arrived as follows:

Day of Week	Day No.	No. Defects
Monday	1	11
Tuesday	2	10
Wednesday	3	7
Thursday	4	7
Friday	5	4

This data is graphed in **Figure 13**.

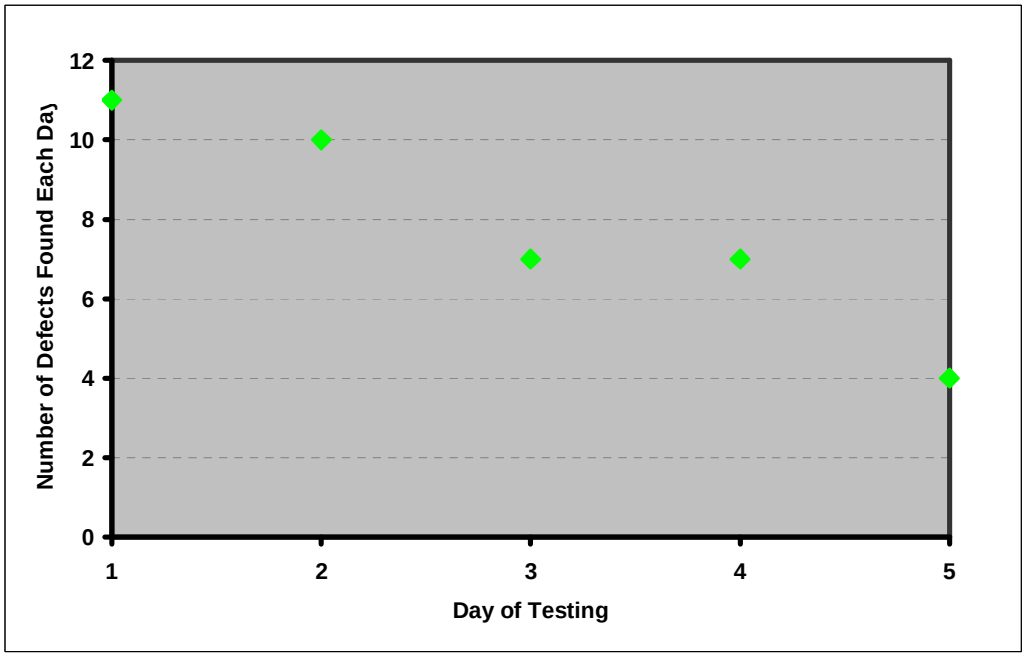

Figure 13. Defects Found Per Day

As presented in **Figure 13** (the most common way of viewing defect arrival rate) the data look pretty scarce. But there is a definite trend toward fewer defects being found each day. We can try fitting a decaying exponential curve to the data of the form

$$\dot{D} = ae^{-bt} \tag{1}$$

where

t is time in days,

a and *b* are arbitrary positive constants to be determined by the curve fit,

and where

$$\dot{D} = \frac{dD}{dt} \tag{2}$$

is the defect arrival rate where *D* is the sequential defect number at any time *t*. **Figure 14** shows the data from **Figure 13** with the curve fit and the resulting equation added. The curve fit estimates the constants *a* and *b* as follows:

$$\bar{a} \cong 15.024$$

$$\bar{b} \cong 0.238$$

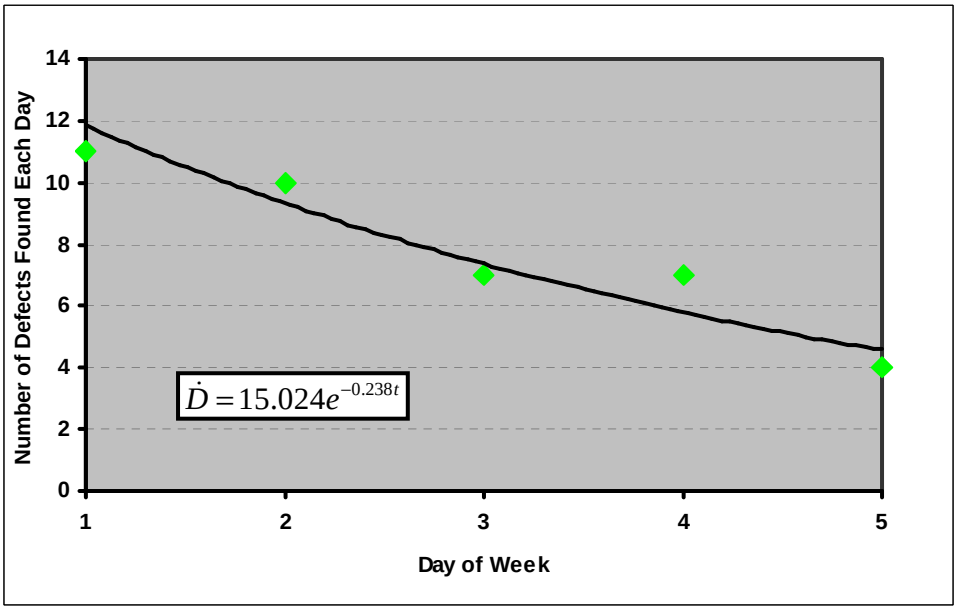

Figure 14. Small Project Defect Arrival Rate

To estimate the number of remaining defects using this equation, we integrate it from 5 days to infinity. We'll skip the math here for the sake of brevity, but the estimated number of undiscovered defects is 19. This is actually not a bad estimate (we generated the data and therefore know that there are actually 21 defects remaining at this point).

There is one situation to which this technique *cannot* be applied: If there is ever a unit of time in which no defects are found, then the curve fit attempt will fail completely and no solution will be possible.[5] For such a case, we'll need another approach.

5.2.1.3. Viewing the Same Data Differently

Another way to look at the same data is as test time between defect reports. This alternate view is particularly useful for very short test cycles or low defect counts (as in this example) or where there are periods of time where no defects are found. **Figure 15** illustrates our short test project data as time between defect reports.

At first glance, this view of the data does not look very helpful. There are a lot more data points, but there is also a lot more noise in the data. But before we give up, let's look at how we might find a useful solution.

[5] The reason for this is that the solution for the curve fit is actually performed in natural logarithm space and the natural logarithm of 0 is undefined (negative infinity).

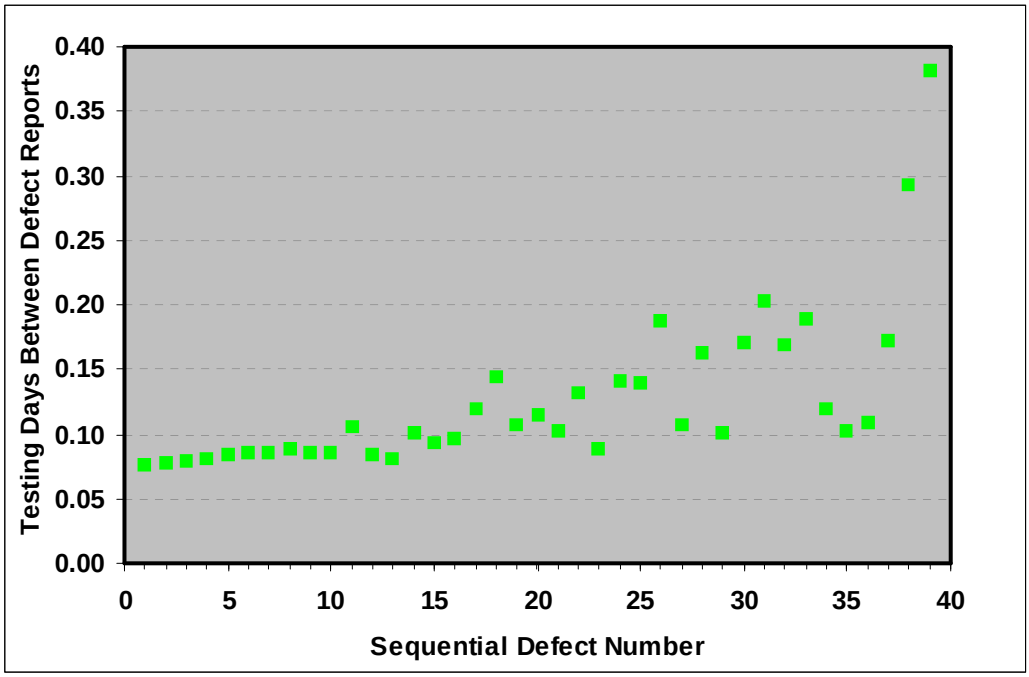

Figure 15. Days Between Defect Reports

We can start with equation (1) (estimated defect arrival rate) and can solve for the corresponding time between defects. If the defect arrival rate is given by equation (1), then the time between defects must be

$$\Delta t(D) = \frac{1}{a - bD} \tag{3}$$

where *a* and *b* are the same constants as in equation (1) and where *D* can be thought of as the sequential defect number (the derivation of equation (3) from equation (1) is left to the reader).

We can now estimate *a* and *b* again by fitting a curve of the form given in equation (3) to the data in **Figure 15**. Doing so gives us

$$\bar{a} \cong 14.21$$

$$\bar{b} \cong 0.231$$

Note the similarity between this estimate of *a* and *b* and the previous estimate. **Figure 16** shows the curve fit and the resulting equation.

Since this *a* and *b* are the same constants as in equation (1), we can plug them directly back into equation (1) to get our alternate estimate of the defect arrival rate:

$$\bar{\dot{D}} = 14.21e^{-0.231t} \tag{4}$$

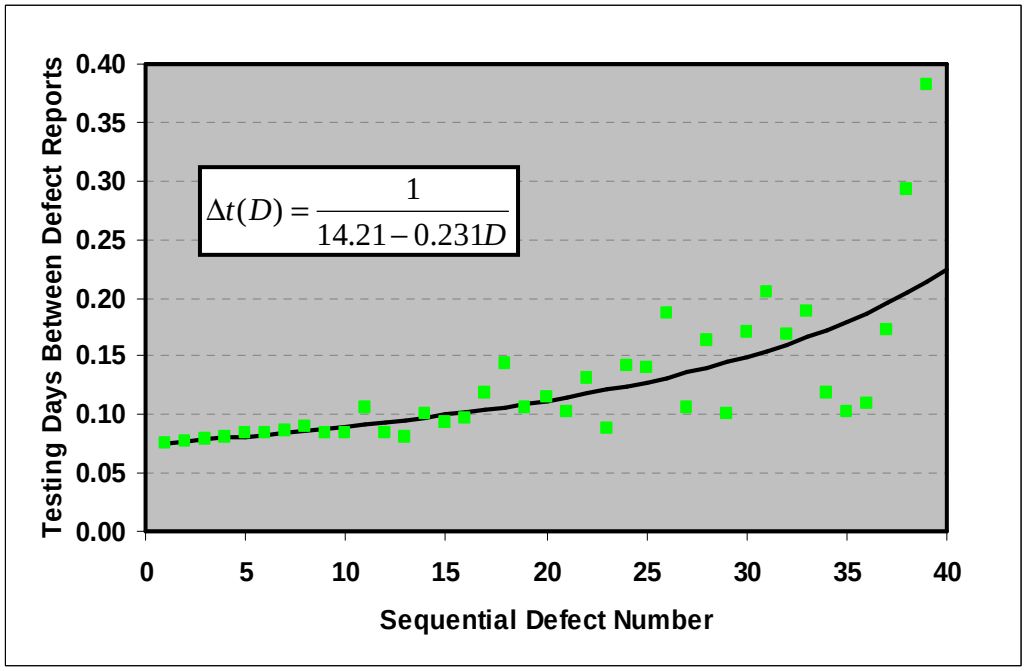

Figure 16. Days Between Defect Reports with Curve Fit

5.2.1.4. Making Decisions Using Defect Arrival Analysis

Of course, the point is not that one of these visualization techniques is inherently better than the other. Rather, the point is that where one is unhelpful, the other is an available option.

How can this analysis help us? We already know that we have found 39 defects in one week of testing. Integrating equation (4) from 5 days to infinity tells us that there are about 21 defects remaining (recall that we estimated 19 remaining defects using the decaying exponential fit). In other words, about 1/3 of all of the defects entering system test still remain undiscovered. Our analysis tells us that it will take a ridiculous amount of time to find ***all*** of the remaining defects (plug D=60 into equation 3 with our estimates of *a* and *b* to see how ridiculous). That information is *not* very helpful. But how many bugs could we find if we spend another week testing? To estimate this, we integrate equation (4) from 5 to 10 days.

Doing so tells us that we should discover an additional 13 defects in another week of testing, leaving only 7. This being the case, we will certainly keep testing. Then we can always wait until the end of the second week to decide whether to continue testing beyond that time. But what might we accomplish by adding test resources to the effort? We can estimate the effect of doubling our test resources by integrating

from 5 to 15 days rather than from 5 to 10 days.[6] We estimate that doing so should result in the discovery of 17 more defects in the next week, leaving only 3.

There is one final caveat concerning defect arrival rate analysis. The primary problem with the defect arrival rate metric is the variation in the amount of actual testing per unit calendar time. If we simply use calendar time for our analysis while our test staff was diverted to training or support for two days – or if someone was out sick for half a day – then the arrival rate data can be muddied to the point of uselessness. Some defect arrival rates look like textbook examples of familiar curves when no attempt is made to calibrate for actual test time. Others are quite literally useless unless such a correction is made. Still others are simply useless for a host of other unavoidable reasons.

But if we can get our test staff to record when they actually start and stop testing, then we can reduce the noise in the data. This significantly improves the odds of getting useful information from this metric. This will have the bonus side effect of making our test engineers aware of how much time they are spending on activities other than testing and should increase our test productivity.[7]

5.2.2. Test Team Performance

The suggested metric for gauging test team performance is the percentage of defects found after product release. This metric should be assembled by module or feature, and should ultimately be correlated to test engineer areas of responsibility. Remember, the point of testing is not to confirm that a product is great (although testing might do that), but to find defects before the customer finds them. If this metric is used, it must somehow be calibrated against the software functionality covered by each tester and against the absolute difficulty of performing tests in each of the various software areas.

The same arguments and cautions that were covered in **Section 5.1** (i.e., concerning metrics gauging individual performance) apply equally well here. If we doubt our ability to account for coverage and difficulty among our individual testers, consider this: the alternative is to ignore data that is readily available and guess at *all* aspects of a tester's performance. Most competent engineers prefer that their supervisors gather any and all objective data that is available.

---

[6] I.e., adding two weeks assuming constant resources is equivalent to adding one week of doubled resources.

[7] As a test engineer, I will say without hesitation that I am happy to take this extra step in order to improve the usefulness of the defect arrival rate data.

5.3. Defect Correction Metrics

Of all of the ways to spend time and money on quality, correction of known defects is undoubtedly the simplest way to get a quick return on investment. Does this mean that we should correct every known defect regardless of its severity before we can release? Not necessarily, just as we would not continue to test until the software is "perfect."[8] But it is obvious that fixing a known defect is almost always less costly than finding and then fixing an unknown defect.

A quick review of **Sections 2.4** and **2.6** suggests that the best source of information concerning the cost of released software defects is our support organizations. Beyond that, customers, competitors, and industry peers provide valuable feedback, even if it can sting sometimes. Without even attempting to develop a mathematical or statistical solution to this problem, we can ask ourselves several questions:

- ✓ What percentage of call center's time is spent on calls about actual defects in software (versus customer questions, confusion, etc.)?
- ✓ What percentage of field engineer time is spent on explaining or working around actual defects in the software (versus nominal installation assistance, etc.)?
- ✓ Is there a subset of defects that seem to generate an inordinate number of calls or a high level of surprise, anger or frustration among customers?
- ✓ Have customers postponed initial or follow-on purchases or threatened to do so until some bugs are fixed? Do we typically have such a "punch list" with new customers?
- ✓ If our product was reviewed by an industry publication, how was it rated for reliability?
- ✓ How do our technical and field support teams rate the reliability of our product?
- ✓ Do we include a list of known problems with every release of our product? If so, do they describe serious problems or require acrobatic workarounds or are they minor irritants that are easily ignored?
- ✓ Does our test and/or development staff often get called upon to help solve support issues?
- ✓ Have our customers begun to skip the call center and call the engineering staff – or worse yet, the CEO – about their problems?

[8] By the way, finding the 59th defect in our previous example is estimated to take an additional 36 test-hours after the 58th defect is found. Finding the 60th defect would take an infinite amount of testing according to our estimate. These numbers are only estimates for a fabricated example, but the law of diminishing returns for testing is very real.

- ✓ Are we having any difficulty getting insurance in field with high liability risks? What about certifications? Agency approval?
- ✓ When we demonstrate our product, are there some features we intentionally avoid because of their usability or reliability?

There are more such probing questions that could be asked, but this list should clearly illustrate the point. If we aren't asking these questions at all, then we are almost certainly underspending on quality. If we answer them objectively, then the answers will tell us if we have a quality issue.

Once we have the answers to these questions, the next place to look is in our defect tracking database. Did we release with several known defects? Were any of them marked with a severity level greater than minor? If so, we'll probably see the effects in the answers to the questions above. Even if we released with only minor defects, we may see them showing up in our tech support costs (see **Section 3.3** above). In that case, we will probably save money by fixing them.

Another bit of data that we can collect and analyze in this area concerns implicit feature requests and documentation defects. Once we have analyzed our tech support expenditures on defects, let's go back and review them again. Are there certain features or procedures that frequently seem to confuse customers? Are there feature requests that have come up repeatedly? Even if such requests are not identical, there may be groups of them that point to a deficiency in some area of the software, help system, and/or documentation. Any such areas that are identified should be slated for enhancement as soon as possible. Doing so will improve our customers' experience and again reduce our support costs.

One final note on defect correction… Remember that a defect can have effects beyond those that are immediately obvious. Moreover, defects can mask or exacerbate each other. The more defects there are in a product, the greater this negative interaction between defects. Allowing defects to accumulate in a product over multiple releases (which is what will happen if the decision is made in each release not to fix all of defects that were found) will eventually result in an unmanageable defect database and product of diminishing quality.

6. Conclusions and Recommendations

So how do we analyze the metrics and use them to effectively accomplish the simultaneous optimization of defect prevention, defect detection, and defect correction? The bottom line is our comfort level with the answers to the questions in **Section 5.3**. If we at least make the effort to examine our support efforts and our testing efforts, we are more likely to move toward the optimal point. If we are happy with our answers to all of the questions in **Section 5.3**

and we spend a good deal of money on defect reduction efforts, then we may be overspending on quality.[9]

6.1. Concerning Test Automation

One final recommendation concerning defect prevention deals not so much with metrics, but with the means of preventing defects. I have never actually witnessed a development shop that clearly and adequately defined "unit test." It would seem that this should be taught in software engineering programs, and maybe it is, but we will probably dramatically improve our defect prevention performance by simply offering our staff a clear definition of a unit test. Consider the following proposed definition:

A software unit test is a white-box test written and executed by a developer against his/her own code with the intent of exercising it as completely as possible and in small units. The unit test should focus on exercising the software as software, not necessarily as an integrated product.

The reason for this definition is very simple. Very little value is added if developers simply try to predict what system testers will do and beat them to the punch. Doing so will not be terribly effective (it takes an entirely different temperament to be a good tester than to be a good developer). But even if it is perfectly effective, it is nearly a total duplication of effort! It is hard to imagine anything ***less*** optimal.

Most system tests do not lend themselves readily to automation [Kaner, et. al, 2002]. On the other hand, almost all unit testing can and should be automated. Barry Mullan, at the Year 2000 Pacific Northwest Software Quality Conference, presented a paper called "The Future of Developer Testing for Java" [Mullan, 2000]. In this paper (most of which easily generalizes beyond Java) he demonstrated that the largest gains to be realized from test automation are in unit testing. He is not the only person preaching this message.

Furthermore, rigorous unit testing is much more likely to uncover those really nightmarish bugs like errant pointers, memory time bombs, memory leaks, etc. that are often difficult if not impossible to reproduce in a fully integrated system.

6.2. Prioritizing Quality Improvement Efforts

Is there an order of importance to the above recommendations for eliminating defects? To some extent, there is. To get at the answer to this question, let's recall a few points:

[9] If you are in this position, give me a call. I'd love to meet you and talk about how you got to this point.

- ✓ According to TARP [Goodman, 1999] it costs anywhere between 2 and 20 times as much to gain a new customer as to retain an existing customer. So we are well advised to make our existing customers happy before worrying about new ones.
- ✓ It is a lot cheaper to fix a known defect than to find and fix an unknown one.
- ✓ We can't fix defects that we don't know about.
- ✓ People are a lot more important than processes or tools (refer back to **Section 4**).
- ✓ Automation efforts are best directed at white box unit testing.
- ✓ Duplication of testing effort between system test and development test is a waste of resources.
- ✓ Having and using objective data is better than not having or not using objective data.

The relative importance of our recommendations now becomes apparent. Here is the suggested order of priority:

1. Hire/promote managers and development and testing staff carefully. If the goal is quality, then the development staff ***must*** be on board.
2. Gather objective data. We can't address problems that we don't know about.
3. Measure and reward quality ***results*** (not just "efforts").
4. With very few exceptions, fix all known defects.
5. Prevent defects whenever possible during development (No. 3 above and a good definition of a unit test is a great start).
6. Find the defects that can't be prevented and fix them when they are found.
7. Assess product quality before release by testing deliberately and using latent defect estimates.
8. If test automation is used, then focus on automating unit tests ***before*** system tests.

Note that *none* of the above recommendations include recommendations concerning process. If, after implementing some or all of these recommendations, we want to standardize on them, we can certainly do so. But the point of this paper is that anyone can make great strides toward better quality and higher profits without implementing someone else's onerous process.

Of course, there is one tacit assumption that pervades the entire discussion and that is that the data discussed throughout this paper is actually being

collected. If it is not being collected, then, of course, doing so is the obvious place to start.

Appendix A – Impact of Delay on Revenue Potential

Any new product has revenue potential that at first grows as the market approaches its "sweet spot" in demand for the product. At times prior to this sweet spot, the market demand has simply not fully crystallized.

For example, although there is a sizeable market for 200+ GB hard drives today, the market for such drives 20 years ago would have been a tiny fraction of today's market because if there was enough digitized data in the world to fill one of these drives in 1985, there certainly was not in the average household!

Then, as time marches on, competitors will enter the field and will begin to compete for a limited number of customers. They will find ways of producing better competing products and selling them at reduced prices. If our product is not likewise improved, it will see rapidly diminishing marketability. This market potential versus time is illustrated in **Figure A1**.

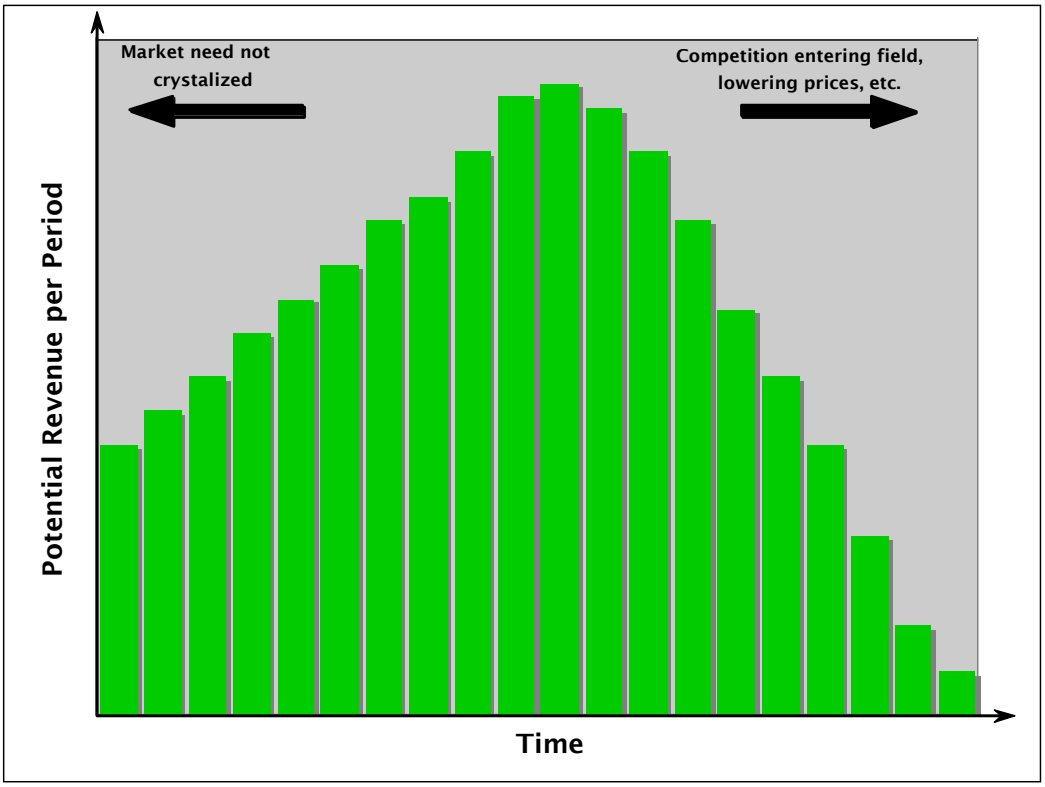

Figure A1. Potential Revenue vs. Time for a Stagnant Product at a Given Quality Level

But when deciding when to release a product, we are not as interested in a quarter-by quarter look at revenue potential as we are in the lost opportunity costs due to delaying the release. A more useful way to present this relationship is as *potential total revenue* for a product given a specific release

date. Such a curve would be obtained by starting at each point on the graph in **Figure A1** and integrating to infinity. **Figure A2** illustrates this result for several quality levels.

Figure A2 illustrates that the potential total revenue for a product is highest when the product is released as early as possible and that this potential begins to decay at an increasing rate at some point – consistent with increased competitive pressures. It also shows that the potential total revenue increases with the quality of a product, but that even for high quality products, potential revenue falls to an unprofitable level eventually. **Figure A2** expresses yet another thing that we all know instinctively: quality is great, but not if it causes us to miss the window of opportunity when the market presents it.

Figure A2. Potential Total Revenue vs. Release Date for Several Quality Levels

So to recap the costs of high quality, **Figure 5** illustrates that beyond a certain point, incremental increases in quality get very expensive. And **Figure A2** warns us against letting our testing and debugging take too long.

References

Garmus & Herron, " Function Point Analysis: Measurement Practices for Successful Software Projects," Addison-Wesley Information Technology Series, December 15, 2000.

Goodman, John, "Basic Facts on Customer Complaint Behavior and the Impact of Service on the Bottom Line," Technical Assistance Research Programs (TARP), June 1999.

Hammond, John S., "Smart Choices: A Practical Guide to Making Better Decisions," Broadway, March 5, 2002.

Hoffman, Douglas, "The Darker Side of Metrics," Pacific Northwest Software Quality Conference, 2000.

Kan, Stephen H., "Metrics and Models in Software Quality Engineering," Addison Wesley, 2003.

Kaner, Bach, and Pettichord, "Lessons Learned in Software Testing," John Wiley & Sons, 2002.

LeBoeuf, Michael, "How to Win Customers and Keep Them for Life," Berkley, 1987.

Mullan, Barry, "The Future of Developer Testing for Java," Pacific Northwest Software Quality Conference, 2000.

Schneier, Bruce, "Beyond Fear," Springer, July 28, 2003.