Software Quality

Let's Discuss This "Can of Worms"

Robert Mattson
9545 Delphi Road S.W.
Olympia, Wa 98502

Introduction

I assume everyone developing software wants it to be of the "highest quality." Recently, I've focused my attention on this goal, what it means and how to move toward achieving it. This has led me to some insights about some key concepts that affect the achievement of the goal "high quality software." Further, I've developed a practical technique for "defining" and "measuring" quality. I wish I had understood these concepts and this technique years ago. It would have improved a lot of software with which I've been associated. I hope the following discussion provides "food for thought" and a technique you can apply. My goal is to contribute something that will be used to improve the "quality" of software.

Why Care About Quality?

It seems that our lives are filled with the "reasons" for producing high quality products. Publications, television and people appear to be constantly communicating the reasons for and exhortation to the production of high quality.

The storyline is the same related to systems/software. There are numerous books and technical articles which tell of the high cost of "poor quality" and the savings from "high quality" systems work. The cost of poor quality in terms of system development cost, development time, user dissatisfaction and software maintenance is today staggering, if one extrapolates from the published numbers.

Yet, what would we find if we could "magically" measure the "quality" of all the software produced this last year with that of ten years ago? Have we made significant gains in the "quality" of software? I imagine that we could find some examples of higher quality software than that of ten years ago. But what about the average piece of software from last year? I'm afraid I'd have to say that, in my

opinion, on average it isn't significantly better than ten years ago. The literature tends to support this opinion.

I heard a well traveled "guru" of this business recently state that "the majority of systems people haven't read a systems related book since they got out of school." If this is truly the case, is it any wonder our software isn't of the highest quality or at least getting better.

If this is the case, why is it? Partly, I believe it stems from the "cultural attitudes" of the people developing the software. Systems workers in this regard have been influenced by the same "cultural attitudes" as the "blue collar" worker or any other worker in our country. The average worker in the U.S. today would not rate "producing what they produce at the highest quality" as even close to their most important goal in life or more significantly "at work." They might pay lip service to it but you probably wouldn't find it rated very high if you could tap their true value system.

There are other reasons that software is not of the highest quality. Many times we are not asked to produce high quality. Or we are asked but not given the resources. Or we see that what is rewarded is not high quality, so we "play the game." There are numerous other "environmental" causes.

In spite of the gloom, there is a brighter view. There are people and groups of people who greatly value the goal of excellence and believe in producing high quality products including software. These people want to and, in some cases, are producing top quality software. Others want to increase the quality but aren't sure how. Hopefully, it is to this latter group that the rest of the concepts and techniques outlined will be of the greatest benefit.

Why is "Software Quality" a "Can of Worms"?

This issue of "software quality" is a "can of worms"... because there seems to be so many rigid "beliefs" about what is "True". As I started to explore this issue, I asked a number of people to tell me what makes for "high quality software". The answers were anything but in agreement. Some people said "maintainability", some "efficiency", some "meeting the users expectations". Some even talked in terms of the fact that there are "probably a number of things".

But the general trend was definitely toward one or two strongly felt parameters.

Additionally, this area is one that has a lot of "nerves" attached to it. If you want verification of the sensitivity of the subject, try leading or doing some code/software or design "walkthrus." Such discussions can easily and quickly degenerate into heated arguments if someone happens to touch another persons pet quality belief (nerve).

The literature I found didn't seem to add much enlightenment to the issue of software quality. There seem to be a number of "camps" or "approaches." Probably the most visible approaches are the "better testing" and "structured analysis" ones. As an aside, what is most surprising is how little one can find on the subject that goes beyond vague generalities.

A significant consequence of this disagreement and lack of attention about what constitutes software quality is either lower software quality or at least not much improvement. Why is that? Well first of all, two people with rigidly held opposing views seldom talk or learn from each other. And we know, when people can't or don't discuss an issue they seldom can learn much from each other. It follows that if we aren't learning we aren't improving. Additionally, people pay lip service to "quality" without realizing the fuzziness of their meaning. This lack of a clear "model" and definition many times results in each person "doing their own thing."

So, what can we do to get control of this "can of worms?" I believe the first step is to look at the whole issue in a new way. The second step is to apply the new techniques that flow from this new view.

Software Quality Viewed as Quality Parameters and Values

The key to understanding software quality is to apply a new "model" to it. This "Mattson Model of Quality" starts with an understanding of the multi-dimensionality of the "quality parameters" related to software. Additionally, one needs to understand the concept of "values", or the "weight" placed on a parameter. Then to complete the model we need to understand about the differences between the "judges." Finally, for greatest benefit, we need a technique to apply the "model." Let's take each in turn.

Quality Parameters

What do I mean by "quality parameters" or "QPs" if you will? "QP"s is a description for all the categories or areas of measurement by which one might judge software quality. For example a common QP is "speed". Another common QP is "documentation." Following is a reasonable list of the QPs for a "program".

Program Quality Parameters:

- Functional Specifications
- Suitability / Job Effectiveness
- Speed / Responsiveness
- Resource Impact
- Robustness / Forgivingness
- Adaptability / Flexibility
- User Acceptance / Satisfaction
- Business Cost Effectiveness
- User Independence / Support
- User Documentation
- Ease of Learning
- Ease of Use / User Efficiency
- Implementation / Installation
- User Interface Uniformity
- Development Task Management
- Cost to Develop
- Time to Develop
- Test Plan / Testing
- Technical Review / Walkthrus
- Defects / "Bugs"
- Maintenance Time/Cost
- Maintainability
- System/Internal Documentation
- Adherence to Standards
- Integration

The most important thing to notice is that there are many different "quality parameters" by which we can measure software. Conversely, there is not just a single measure. In other words, "bugs/defects per line of code" or "structured code" or "maintainability" are only one of many possible QPs. Notice also that each of these QPs has itself potentially "sub-QP". In other words, "documentation" might be divided into "user" and "system" documentation.

Part of the challenge in this approach is coming up with a list of Qp that is comprehensive without being unmanageable.

How then do we measure each QP? Some QPs such as "speed" may have quantitative measures. But for most QPs there is no clear quantitative measurement. For example what is the quantitative measurement for the "quality" of "system documentation". The strategy that seems to work best in these cases is to discuss the parameter in terms of three levels: unacceptable, ok, excellent. Thus, taking our example of the QP "system documentation" we can usually define what is unacceptable, what would be ok, and what would be excellent. Many times it helps to use examples or references to standards to communicate these levels.

Ah, but you say, what is not acceptable "speed" for one program is excellent for another. That is why there is no such thing as THE quality way, technique or point. Rather, excellence must be defined for each situation. That is why we must discuss and come to agreement as to what the various levels are for each particular piece of software (for example each program). This discussion helps define the different views of quality for each QP. This is very important. We assume too often, I'm afraid, that each party involved in the development of software have the same measure of excellent quality for any particular QP.

How does one start to apply this concept? Refer now to Exhibit #1. This is a "Software Quality Form" that is used for documenting "quality" for a program. This form shows a reasonable set of QPs for writing a program and has room for documenting key points related to the different quality levels. See Exhibit #2 for what this form would look like when we fill out the "Quality Level" fields.

This form is filled out before a program is developed. Let's focus for the moment of the columns labeled "Quality Parameters", "Unacceptable", "Ok" and "Excellent". The person who is responsible for doing a program fills out the "Quality Level" columns on the "standard" program version of the form. In addition, they add any other QPs that might be of special significance. The description of the levels for each QP is discussed with the persons project leader and/or supervisor. Some of it can be discussed with the user. The purpose of the discussion is to get an agreed understanding between the parties as to the definition of levels of quality for each QP for this program.

You may have noticed some other columns on the form. To understand the reason for these and their use we need to address the next dimension of the problem... that of "value" ratings.

The Concept of "Values".

I've describe above how to establish the QPs and specific quality levels by which we can judge the software. Now we need to discuss the concept of how much "value" we place on each.

The concept of "value" has to do with the "weighting" we put on achieving the "excellent" or "ok" level of "quality" on any QP for this program. In this technique a "value" is placed on the achieving of the "ok" and "excellent" level of each QP defined. These are "relative" values. In other words giving one QP level a 10 and another a 20 means that achieving the second(20 pt value) is twice as important as achieving the first (10 pt value).

Why do we need/want to do this? First, because sometimes the achieving of an excellent level in two QPs are counter to each other. For example code size and user flexibility are usually mutually exclusive. In this case the developer needs to know which QP to emphasize. Other times the QPs are not counter to each other but there is simply not enough "time" or "resources/dollars" to achieve excellence in all areas. In this latter situation, we must know where to place our emphasis.

The numeric "values" also give us more information than such statements as "speed is important" or "I want us to emphasize maintainability". The relative values of the numbers provide us with much better information on "how much" we want to emphasize one QP over another.

Let me give you an example. I want to develop a program that makes a "fix" to my database and I know I'll only use it one time. This database has 100 million records. I define the QP "speed" in terms of unacceptable, ok, and excellent. I define the QP "system documentation" in terms of its three levels. Note: I'm only using two QPs for clarity but one would have "defined" many QPs and placed "value" on achieving their different levels of quality. Here's how those "values" might look.

	----- Quality Level -----	
	"OK"	"Excellent"
Speed	10	50
System Documentation	5	10

Now if I had to make a choice between achieving the excellent level in "speed" or "documentation" which would I want? What if chosing one means the other QP will only be achieved at the "OK" level?

I believe software professionals make these kind of trade-offs in their head. At the same time I've found that far too many times the trade-offs made by one person are in disagreement with those another might have made. This model/technique allows for the good communication of the possible trade-offs. This model/technique will help avoid the frustration, conflict, wasted dollars and effort associated with making the "wrong" trade-offs.

Implied in this technique is an associated rule. The rule is that unless agreed to by the "players" in the process (i.e. user, manager, project leader, programmer team associates, etc) it is not acceptable to achieve the "unacceptable" level for any QP. This is true even if this QP has little value. If the "significant others" agree to achieving the "unacceptable" level then it is alright to do this. Note, this new agreement has really just been to define "ok" to mean whatever "unacceptable" had been. A special case is when a QP is valued at zero. This means that no value is placed on this parameter and anything is probably acceptable.

Now, the reason for the "value" columns on the Software Quality Form is clear. This is where the "values" can be "set" for a piece of software. The doer assigns "values" after filling out the description of the quality levels. These are also discussed with the supervisor and/or leader and where appropriate with the user. All parties come to agreement on what the relative values of each achievement for each QP will be. Notice, there is not a value column for the "Unacceptable" level; this is because there is NO value in doing this!

A Word About "Judges"

There are a number of people who will ultimately judge the "quality" of a piece of software. These "judges" include the "builder", the builder's associates, the users, the builder's manager, "outsiders", etc. We may not wish to have it judged but it is a "fact of life."

It may be obvious by now, two different people ("judges") would probably do the following differently if asked to do it "separately".

1) Describe the QPs by which to judge software Quality.
2) Describe what "unacceptable", "ok" and "excellent" levels of achievement are for a particular QP for a particular piece of software.
3) Rate the relative importance (values) of the QPs and their levels for the piece of software.

The important thing to understand is that this happens all the time in the "real world." What confounds us many times is that we unconsciously assume that the result of the three steps above is same for both parties. Then we wonder why our supervisor is less than happy about the "excellent" work we just completed. Or if we are a supervisor we wonder about the "competence" of our employee who completed such "poor quality" work. Further, because we have no formal and written quality specification we have to rely solely on our memories of whatever discussions we might have had about this program. If it is obvious that the parties "disagree" on the "quality" of the software, relying on memories will usually not lead to a productive and positive resolution. Rather, what usually happens is either no discussion takes place or one does but it results in emotional exchanges.

It is very enlightening to take a Software Quality Form for which we've completed the quality level specifications only and give it to all "interested" parties. It will be enlightening to see how the different people will value the different QPs and levels of quality.

Is there a better way to deal with the differences that so commonly arise currently? Yes, the better way is to apply the "model/technique" I've outlined.

Using the Technique --- The Software Quality Form

You've already been introduced to the Software Quality Form. The use of it is fairly straight forward but there are a few additional steps in its use. Here are the steps:

1) Discuss, decide and list the QPs that you want to use in the judging of this type of software.
2) Discuss, decide and document what each level of achievement for a QP would be.
3) Discuss, "agree on" and document the relative "values" to place on each level of achievement for each QP.
4) Add up the Excellent column of "values"
5) Have the builder and one or more of the other "judges" rate each QP for achievement of the goal.
6) Total the rating points for each "judge" and calculate the percent of "Excellent" achieved.
7) Discuss the difference in ratings and the ways to increase the "excellence" (percentage) next time.

It sounds simple, doesn't it? And really it is! It does take a little time. But the time is very small compared to most software development efforts. The second time it is done will be faster than the first, the third time faster than the second, and so on. The QPs will tend to be established and not changed for each new program. What will change is the definition of "Ok" and "Excellent" levels of achievement as well as the value placed on each. But even there, you'll see a lot of re-use of descriptions and similar values for somewhat similar software.

The use of this form will improve the quality of software. Why? First, just using this technique will be helpful in discovering how often "quality" is poorly defined. Everyday, you'll see the "word" quality used as if its meaning was unmistakable. Having a better understanding of the nature of "quality" will allow one to handle this fact. Secondly, people will spend time thinking about what excellence is. Consequently, they will have clearer goals for achieving quality. Therefore, they will be much more likely to develop the features that result in "quality." Conversely, less time will be spent doing the wrong thing or emphasizing the wrong QP.

Conclusions

There is no "absolute", "universal", and "always" way to measure the level of achieved software quality. Neither, however, is quality only in the eye of the beholder. There is a middle ground. The quality of most software can be judged on the basis of some "quality parameters" or QPs. For each QP there are unacceptable, ok and excellent levels of achievement. Each of these levels has a relative importance to the various "judges" of the software. The QPs, levels, and values can be established for any piece of software. The product produced can then be evaluated against these pre-established measures. We can then calculate the "level of excellence" achieved as a percentage of total excellence conceived. This model/technique makes significant improvements over the simplistic and/or poorly defined methods commonly employed.

This conceptual model can be applied to the "real world". The use of the Software Quality Form is an efficient way to do this. The benefits of applying this model are easily worth the time/cost spent. "Quality" will improve because we will have a clearer picture of what it is, what we want and whether we are attaining it!

Software Quality Form

System Name: _____ Est. Time: _____
Program Name: _____ Actual Time: _____
Date: / / Assigned To: Reviewed By:

Quality Parameters	Quality Level					Rating	
	Unacceptable	OK	Value	Excellent	Value	Doer	Rev.
Functional Specifications							
Suitability Job Effectiveness							
Speed Responsiveness							
Resource Impact Overhead							
Robustness Forgivingness							
Adaptability Flexibility							
User Acceptance Satisfaction							
Business Cost Effectiveness							
User Independence Support Required							
User Documentation							
Ease of Learning							
Ease of Use User Efficiency							
Implementation Installation							
User Interface Uniformity							
Development Task Management							
Cost to Develop							
Time to Develop							
Test Plan Testing							
Technical Review Walkthrus							
Defects "Bugs"							
Maintenance Time & Cost							
Maintainability Int. Documentation							
Adherence to Standards							
Integration							
Comments:				TOTALS			
				% of Excellent			

The Goal is Excellent Systems

Exhibit #1

Software Quality Form

System Name: PURCHASING System Est. Time: 60 hrs.
Program Name: PWS008 - INVOICE PROCESSOR Actual Time: 65 hrs.
Date: 4/20/88 Assigned To: S. CARNEGIE Reviewed By: R. MATTSON

Quality Parameters	Quality Level: Unacceptable	OK	Value	Excellent	Value	Rating: Doer	Rev.
Functional Specifications	Miss Req. Spec.	Del. Req. Spec.	20	Del. Req & Some "Nice" Spec's	40	40	40
Suitability Job Effectiveness	Can't Handle Spec. Trx's.	Handle All Spec Trx's With Fixes	15	Handle All Spec Trx's Corr. 1st. Time	20	15	15
Speed Responsiveness	G.T. 5 sec's Trx. Time	2-5 sec's "Design Spec"	5	L.T. 2 sec.	10	5	5
Resource Impact Overhead	Anything like PWS006	Cause no sys. "Problem".	5	Sys. Res. Low	10	5	5
Robustness Forgivingness	Any "Abort"	No Aborts or editable errors.	5	OK + able to correct all errors	10	10	10
Adaptability Flexibility	Can't Handle Known Trx's	Handles only known Trx's	10	Handles New Types Not "Defined"	12	10	10
User Acceptance Satisfaction	User "rejects" or has "Complaints"	User Accepts & only has enhancement req.	5	User Praises it	6	5	5
Business Cost Effectiveness	Takes More time for Trx than manual	Takes Same time but fewer errors	10	Takes Less Time for Trx & fewer errors	20	20	20
User Independence Support Required	D.P. Req For Overide Oper.	Minor yearly tasks	5	No Task, user's System.	10	5	5
User Documentation	No Help File	Help File	2	Better Help File (?) User Reviewed	4	2	2
Ease of Learning	User takes G.T. 1 Day to Learn	L.T. 1 DAY & G.T. 1 HR.	2	User Takes L.T. 1 hour to Learn 90% Level	4	4	4
Ease of Use User Efficiency	User Complains Its HARDER	User Does Complain or Praise	2	User Says its Easier	4	2	4
Implementation Installation	Major Glitch	Minor Glitch	5	No Glitches or Surprises	10	10	5
User Interface Uniformity	Any Major Non-Compl.	Major Compliance (See Memo)	20	OK + Improvement	25	20	20
Development Task Management	Est. & Comp. > 10%	Est. & Comp. L.T. 10%	10	Makes All Est. & Comp.	20	10	10
Cost to Develop	No. Comm. Status. Greater $2000	Comm. on Req. From $1000-2000	10	Takes Resp. for Comm. L.T. $1000	20	10	10
Time to Develop	G.T. 80 hrs.	40-80 hrs	10	L.T. 40 hrs	20	10	10
Test Plan Testing	/	Test Plan & Tested	5	Test Plan Before Code & Walkthru	10	5	5
Technical Review Walkthrus	/	Either Design or Code W.T.	5	Both Design & Code W.T.	10	5	5
Defects "Bugs"	> 2 Defects 1st Wk	1-2 Defects 1st Wk.	5	No Defects 1st Wk.	20	20	20
Maintenance Time & Cost	G.T. 20% Maint Time 1st Six Mths.	L.T. 10% Maint. Time 1st Six Mths	5	No maint. Time & $ First Six mths	20	—	—
Maintainability Int. Documentation	No block Stmp or Para. Cmts.	Block Stmp & Para Cmts.	2	OK + System Diag. & Sys. Narr. & Flowchart	10	2	2
Adherence to Standards	Any maj dev. w/o. Agreement.	No major dev. but minor dev. w/o agree.	2	No dev. w/o agreement	5	5	5
Integration	Any major prob. with int.	No major prob. with integration	5	No Problems for	10	10	10
					320		
				TOTALS	330	230	227
				% of Excellent		70%	69%

Comments:
Trx's = Transactions - See Spec Doc.
GOOD JOB. Let's work together on improvement Prog. R. Steve

The Goal is Excellent Systems

Exhibit #2

Software Quality - Let's Discuss This "Can of Worms"!
0079-12