ISSN 2277-3061

Software Quality: Issues, Concerns and New Directions

Kiran Kumar Reddi[1], S.V. Achuta Rao[2]
[1]Krishna University, Machilipatnam, India.
kirankreddi@gmail.com
[2]Research Scholar, Krishna University, Machilipatnam, India.
achutaraosv@gmail.com

ABSTRACT

Software metrics and quality models have a very important role to play in measurement of software quality. A number of well-known quality models and software metrics are used to build quality software both in industry and in academia. Development of software metrics is an ongoing process with new metrics being continuously tried out. However, during our research on measuring software quality using object oriented design patterns, we faced many issues related to existing software metrics and quality models. For a particular situation of interest, any established metric can be used. If none is found to be appropriate, a new metric can be devised. In this paper, we discuss some of these issues and present our approach to software quality assessment.

Keywords: Project Management, Software Quality, Schedules, testing tools.

Council for Innovative Research
Peer Review Research Publishing System
Journal: INTERNATIONAL JOURNAL OF COMPUTERS & TECHNOLOGY
Vol 11, No. 8
editor@cirworld.com
www.cirworld.com, member.cirworld.com

ISSN 2277-3061

INTRODUCTION

In the context of software engineering, software quality refers to two related but distinct notions that exist wherever quality is defined in a business context: Software functional quality reflects how well it complies with or conforms to a given design, based on functional requirements or specifications. That attribute can also be described as the fitness for purpose of a piece of software or how it compares to competitors in the marketplace as a worthwhile product; Software structural quality refers to how it meets non-functional requirements that support the delivery of the functional requirements, such as robustness or maintainability, the degree to which the software was produced correctly[1]. Structural quality is evaluated through the analysis of the software inner structure, its source code, in effect how its architecture adheres to sound principles of software architecture. In contrast, functional quality is typically enforced and measured through software testing.

A good way to start any inquiry is to define your terms. According to Oxford Dictionary, Quality is a noun meaning "degree of excellence". Excellence is defined as "surpassing merit", Merit as "goodness", and Goodness as "virtue". So what we have here is an ethical issue: Quality is the relative virtue of a thing, compared to alternatives. Your software has quality to the extent that it provides Value to some living, breathing people with choices and options. If another program solves a similar problem in a way that the person values more, it has higher quality [2, 3]. Most of these things you should really already be doing and if you're not then I'd suggest starting to do so right away. Poor quality is not an inevitable attribute of software. It results from known causes. It can be predicted and controlled, but only if its causes are understood and addressed. With more critical business processes being implemented in software, quality problems are a primary business risk. I'll discuss five primary causes of poor software quality and How to mitigate their damaging effects using methods other than brute testing [1].

SOFTWARE QUALITY PRINCIPLES

1. Properly managed quality programs reduce total program cost, increase business value and quality of delivered products, and shorten development times [3, 4, 5].

1.1. If cost or development times increase, the quality program is not being properly implemented.

1.2. The size of a product, including periodic reevaluation of size as changes occur, must be estimated and tracked.

1.3. Schedules, budgets, and quality commitments must be mutually consistent and based on sound historical data and estimating methods.

1.4. The development approach must be consistent with the rate of change in requirements.

2. To get quality work, the customer must demand it.

2.1. Attributes that define quality for a software product must be stated in measurable terms and formally agreed to between developers and customers as part of the contract. Any instance of deviation from a specified attribute is a defect.

2.2.The contract shall specify the agreed upon quality level, stated in terms of the acceptable quantity or ratio of deviations (defects) in the delivered product.

3. The developers must feel personally responsible for the quality of the products they produce.

3.1. The development teams must plan their own work and negotiate their commitments with management and the customer.

3.2. Software managers must provide appropriate training for developers. A developer is anyone who produces a part of the product, be it a designer, documenter, coder, or systems designer.

4. For the proper management of software development projects, the development teams themselves must plan, measure, and control the work.

4.1. Project teams must have knowledge and experience in the relevant technologies and applications domains commensurate with project size and other risk factors.

4.2. Removal yield at each step and in total pre-delivery must be measured.

4.3. Effort associated with each activity must be recorded.

4.4. Defects discovered by each appraisal method must be recorded.

4.5. Measurements must be recorded by those performing the activity and be analyzed by both developers and managers.

5. Software management must recognize and reward quality work.

5.1. Projects must utilize a combination of appraisal methods sufficient to verify the agreed defect levels.

5.2. Managers must use measures to ensure high quality and improve processes.

5.3. Managers must use measurements with due respect for individuals.

* These principles were defined by a group of 13 software quality experts convened by the experts Taz Daughtrey, Carol Dekkers, Gary Gack and their team.

EVALUATION OF QUALITY MODELS

McCall (McCall, Richards & Walters, 1977) introduced his quality model in 1977. According to Pfleeger (2001), it was one of the first published quality models measurable properties that can be evaluated in order to quantify the quality. McCall proposes a subjective grading scheme ranging from 0 (low) to 10 (high). Furthermore, some of the factors and measurable properties like traceability and self-documentation among others are not really definable or even meaningful at an early stage for non-technical stakeholders. This model is not applicable with respect to the criteria outlined in the IEEE Standard for a Software Quality Metrics Methodology for a top to bottom approach to quality engineering. Furthermore, it emphasizes the product perspective of quality to the detriment of the other perspectives. It is therefore not suited as a foundation for Software Quality Engineering according to the stated premises.

Boehm's quality model improves upon the work of McCall and his colleagues (Boehm, Brown, Kaspar, Lipow & MacCleod, 1978). As Figure 2 shows, this quality model loosely retains the factor-measurable property arrangement. However, for Boehm and his colleagues, the prime characteristic of quality is what they define as “general utility”. According to Pfleeger (2001), this is an assertion that first and foremost, a software system must be useful to be considered a quality system. For Boehm, general utility is composed of as-is utility, maintainability and portability (Boehm et al., 1976):

- How well (easily, reliably, efficiently) can I use it [software system] as - is?
- How easy it to maintain is (understand, modify, and retest)?
- Can I still use it if I change my environment?

If the semantics of McCall's model are used as a reference, the quality factors could be defined as: Portability, Reliability, Efficiency, Human Engineering, Testability, Understandability and Modifiability. These factors can be decomposed into measurable properties such as Device Independence, Accuracy, Completeness, etc. Portability is somewhat incoherent in this classification as it acts both as a top level component of general utility, and as a factor that possesses measurable attributes [8, 9].

(i) Lack of domain knowledge:

Perhaps the greatest contributor to poor software quality is the unfortunate fact that most developers are not experts in the business domain served by their applications, be it telecommunications, banking, energy, supply chain, retail, or others. Over time they will learn more about the domain, but much of this learning will come through correcting defects caused by their mistaken understanding of the functional requirements. The best way to mitigate this cause is to provide access to domain experts from the business, proactively train developers in the business domain, and conduct peer reviews with those possessing more domain experience.

(ii) Lack of technology knowledge:

Most developers are proficient in several computer languages and technologies. However, modern multi-tier business applications are a complex tangle of many computer languages and different software platforms. These tiers include user interface, business logic, and data management, and they may interact through middleware with enterprise resource systems and legacy applications written in archaic languages. Few developers know all of these languages and technologies, and their incorrect assumptions about how other technologies work is a prime source of the non-functional defects that cause damaging outages, data corruption, and security breaches during operation. The best way to mitigate this cause is to cross-train developers in different application technologies, conduct peer reviews with developers working in other application tiers, and perform static and dynamic analyses of the code.

(iii) Unrealistic schedules:

When developers are forced to sacrifice sound software development practices to ridiculous schedules the results are rarely good. The few successful outcomes are based on heroics that are rarely repeated on future death marches. When working at breakneck pace, stressed developers make more mistakes and have less time to find them. The only way to mitigate these death march travesties is through enforcing strong project management practices. Controlling commitments through planning, tracking progress to identify problems, and controlling endless requirements changes are critical practices for providing a professional environment for software development.

(iv) Badly engineered software:

Two-thirds or more of most software development activity involves changing or enhancing existing code. Studies have shown that half of the time spent modifying existing software is expended trying to figure out what is going on in the code. Unnecessarily complex code is often impenetrable and modifying it leads to numerous mistakes and unanticipated negative side effects. These newly injected defects cause expensive rework and delayed releases. The best way to mitigate this cause is to re-factor critical portions of the code guided by information from architectural and static code analyses.

(v) Poor acquisition practices:

Most large multi-tier applications are built and maintained by distributed teams, some or all of whom may be outsourced from other companies. Consequently, the acquiring organization often has little visibility into or control over the quality of the software they are receiving. For various reasons, CMMI levels have not always guaranteed high quality software deliveries. To mitigate the risks of quality problems in externally supplied software, acquiring managers should implement quality targets in their contracts and a strong quality assurance gate for delivered software. There seems to be three main aspects of a programming language and its implementation which have software quality implications.

(vi) The Design of the Programming Language:

Many of the major issues in programming language design have been extensively discussed over the years, probably ever since the "Structured Programming" was first used, if not earlier, it is fairly well established that the use of a language which actively encourages Structured Programming, such as Module-2 or Ada. All the good aspects of program design relating to program structure, such as modularisation, abstract data types, data hiding, variety of sensible control structures, should be implementable in clear, explicit ways using the language in order to maintain software quality. The conflict between a good language design from the point of view of style and speed of execution of the object programs is not new, although with modern compilers and computer architectures there is probability little difference in the execution speeds possible for the mainstream types of imperative language. However, FORTRAN is still used a great deal for the more mathematical types of problem. Not only because of huge investment in existing software routines, but also due to the comparative ease by which very efficient execution times can be achieved and a long history of usage for these types of problems. Some of the problems affecting software quality can be solved by imposing additional in-house restrictions on the use of the programming languages, one example of this is avoiding the use of dynamic storage allocation and pointer variables where very high integrity software is required. Another example is insisting on the use of explicit declarations for all variables in languages which allow implicit declarations, so as to improve the checking that a compiler can do against accidental misspelling of identifiers.

(vii) Programming Language Standards:

A good paper discussing the effects of programming language standards on software quality has recently been published by Wichmann [7]. There are a number of major decisions that the program standards committees have to make which can have a considerable impact on the type and form of standard produced

Thoughts on the List

The first two causes distinguish between functional and non-functional quality problems, a critical distinction since non-functional defects are not detected as readily during test and their effects are frequently more devastating during operations. The third and fourth causes have been perennial, although the fourth problem is exacerbated by the increase in technologies integrated into modern applications. The final problem is not entirely new, but has grown in effect with growth in outsourcing and packaged software. Just missing this list but deserving of attention are breakdowns in coordination among distributed software teams, a cause that would make the top five in some environments [16]. In well run software organizations testing is not a defect detection activity. Rather, testing should merely verify that the software performs correctly under a wide range of operational conditions. By understanding and addressing the top five causes of defects, quality can be designed in from the start, substantially reducing both the 40% of project effort typically spent on rework and the risks to which software exposes business.

How to improve software Quality with Successful quality Strategies?

Organizations have reached quality levels of few defects per million parts, but these have been with manufacturing and not design or development processes. In the manufacturing context, the repetitive work is performed by machines, and the quality challenge is to consistently and properly follow all of the following steps.

"*Quality Equals Superior Value to the Client*"

The market value of a product is not an intrinsic value, not a "value in itself", hanging in a vacuum. A free market never loses sight of the question: of value to whom?

Visitacion and Gualtieri conclude that software quality is a team sport and everyone needs to play.

"Quality must move beyond the purview of just QA professionals to become an integrated part of the entire software development life cycle to reduce schedule-killing rework, improve user satisfaction, and reduce the risk of untested non-functional requirements such as security and performance," they write. "Managers must make quality measurable and invent all roles on the team to improve it" [9, 10, 11, 12].

(a) Step back and plan

It's often hard not to just jump straight in to coding, especially with a project you are excited about. Try to resist that urge by stepping back and taking a bit of time to think about things before you start typing. Think about the problem are you trying to solve, any difficulties that may arise and come up with a potential solution. Even better than thinking about the problem and solution, write it down somewhere. If you start coding before you've completely got your head around the problem there's a very good chance you will end up with little more than a mess. After all if you start coding a solution before you know what that solution is how are you going to know when you arrive at that solution?

(b) Document before coding

Documentation is another aspect of coding that is too often overlooked. Some developers believe it is not their job to document; others just don't get around to it because of time constraints. Documentation of functions/methods and how particular sections of code are supposed to work can help with debugging and avoid the old "What was I/he/she thinking with this code?" situation. I suggest that whenever you create a new class/function/method (or any other relevant piece of code) create the skeleton of it ONLY, then write the documentation. Only once the documentation is done should you think about implementing the guts of the function. This will ensure documentation doesn't get forgotten and you will have a much clearer picture of what you are trying to achieve before you set out.

(c)Adopt a coding standard and stick to it.

Use consistent indentation, layout, naming conventions etc across all of your code. It not only makes it easier for you and other to read and modify but it will also make it much easier to debug when you come back to look at the code in a few months time. This is easily one of the most important things you can do to improve the quality of your code but is also one of the most forgotten about or ignored.

(d)Write test plans and make sure they are used

A lot of the time testing is performed very minimally or in a haphazard way. This is bad because leaves you open to miss areas while testing or not discovering obvious problems because the tester does know understand how the software will be used. This is unfortunately always a risk but it can be greatly minimized by writing various test plans and ensuring they are actually used. Test plans can be quite complex and extensive or simple use cases that are followed by the tester. Either way they are an excellent way of showing what has been tested and over time they can be expanded to cover more of your software. Similarly to the writing of documentation for each class/function/method when they are created you should additionally create a set of tests for every function as or before it is coded. Again the tests can be expanded over time and are a great way of performing quick regression checks.

(e)Reviews

This is by no means a new concept but it is definitely one that is under-utilised. Developers are often afraid of peer reviews because they don't like their code being criticised and other developers can often be quite harsh. Don't take it personally, it can be very useful to have someone else's eyes spot things you missed and after a few reviews your code will more than likely start to improve in general. In addition to peer reviews, review your own code. Go back and have a quick read of what you've done in 1 week, 1 month, 4 months and 12 months later if you can. You're almost guaranteed to come up with a better solution than you did before. Although you may not be able to implement your new ideas in that old project you may be able to use them in future.

(f) Define Quality to Match Your Needs

***Impact on Quality*:** Meet business requirements; achieve a satisfying user experience.
***Benefit*:** Your ability to achieve quality is improved because the application development team is not charged with unrealistically perfect expectations. Rather, it is chartered with a definition of quality that fits the given time, resource, and budget constraints.
***Relevant Roles*:** Business stakeholders; entire application development team.

(g) Broadcast Simple Quality Metrics

***Impact on quality**:* Reduced defects.

Benefit: Highly visible metrics keep quality top of mind for the entire team and expose when efforts fall short.

Relevant Roles: Entire application development team

(h) Fine-Tune Team/Individual Goals to Include Quality

Impact on Quality: Meet business requirements; achieve a satisfying user experience; reduce defects.
***Benefit*:** Team members perform according to their incentives, making quality improvement part of their goals reinforces desirable behaviour.
Relevant Roles: Management.

(i)Get the Requirements Right Impact on Quality

Meet business requirements; achieve a satisfying user experience;

Benefit: Less rework means less retesting and fewer cycles, which greatly reduce the overall effort.
Relevant Roles: Managers, business analysts, user experience designers, architects.

(j) Test Smarter to Test Less Impact on Quality

educe defects Benefit: A focus on testing the most crucial and at risk areas ensures that they receive the lion's share of test resources and that any bugs that slip through are likely to be confined to the least-important features.

Relevant Roles: Quality assurance, managers.

(k)Design Applications to Lesser Bug Risk Impact on Quality

Reduce defects Benefit: Simpler, cleaner designs result in code that is simpler, cleaner, and easier to test and rework—which means that the code will have fewer bugs and that those bugs will be easier to diagnose and repair.
Relevant Roles: Architects, developers.

(l)Optimize the use of Testing tools Impact on Quality

Reduce defects Benefit: Automation frees resources from mundane testing to focus on the highest-priority tests and increases test cycles' repeatability.

Relevant Roles: Quality assurance, developers.

(m) ***Maintain the entire process under continuous statistical control***

(n)Evaluate the product outputs.

(o)Properly handle all deviations and problems.

(p)Suitably package, distribute or otherwise handle the machine outputs.

(q)Consistently strive to improve all aspects of the production and evaluation process.

While the above points suggest some approaches to consider for software development, they are not directly applicable for human-intensive work such as design and development. However, by considering an analogous approach with people instead of machines, we begin to see how to proceed.

ELEMENTS OF SOFTWARE QUALITY MANAGEMENT

The below steps required to consistently produce quality software are based on the five basic principles of software quality which discussed above in this paper [17, 18].

I. Establish quality policies, goals, and plans.
II. Properly train, coach, and support the developers and their teams.
III. Establish and maintain a requirements quality-management process.
IV. Establish and maintain statistical control of the software engineering process.
V. Review, inspect, and evaluate all product artifacts.
VI. Evaluate all defects for correction and to indentify fix and prevent other similar problems.
VII. Establish and maintain a configuration management and change control system.
VIII. Continually improve the development process.
IX. Organizational structure
X. Responsibilities
XI. Methods
XII. Data Management
XIII. Processes - including purchasing
XIV. Resources - including natural resources and human capital
XV. Customer Satisfaction
XVI. Continuous Improvement
XVII. Product Quality
XVIII. Maintenance
XIX. Sustainability - including efficient resource use and responsible environmental operations
XX. Transparency and independent audit

CONCLUSION

We believe in continually trying to improve skills, knowledge and most of all coding standards. By using the techniques above (as well as others) the quality of software can be improved and over time so will your skills.

REFERENCES

[1] Marcus T. Project management an introduction. URL:http://www.newgrange.org/FTP/pm.ppt

[2] Teamworks. Project management. URL:http://www.vta.spcomm.uiuc.edu/PMT/pmt-ov.html

[3] ISDP. What is project management? URL:http://itprojmngt.8m.net/projman/pm_what.html

[4] Project Management Institute. Project management-a proven process for success. http://www.pmi.org/projectmanagement

[5] TenStep Project Management Process. The value of project management. URL:http://www.tenstep.com/0.0.1%20Home%20-%20Value.htm

[6] Hoffer JA, George JF, Valacich JS. Managing the information systems project. In:*Modern Systems Analysis & Design.* Upper Saddle River, NJ:Prentice Hall, 2002, pp 59-92

[7] ISDP. Role description project manager. URL:http://itprojmngt.8m.net/projman/organization/functions/jd_project_manager.html

[8] Ford J. Workplace conflict: facts and figures. URL:http://mediate.com/workplace/ford1.cfm

[9] Ford J. The training of conflict resolution skills in the workplace. URL:http://www.mediate.com/workplace/workedit3.cfm

[10] Capozzoli TK. Conflict resolution-a key ingredient in successful teams. *Supervision* (60:11), 1999, pp 14-16

[11] Wall JA Jr, Callister RR. Conflict and its management. *Journal of Management*(21:3), 1995, pp 515-558

[12] Dana D. What's a conflict? URL:http://www.mediate.com/articles/dana1.cfm

[13] Klunk SW. Conflict and the dynamic organization. *Hospital Materiel Management Quarterly* (19:2), 1997, pp 37-44

[14] Teamworks. Using team conflicts. URL:http://www.vta.spcomm.uiuc.edu/TCT/tctl1-ov.html

[15] Van Slyke EJ. Resolve conflict, boost creativity *HRMagazine*(44:12), 1999, pp 132-137

[16] Cloke K, Goldsmith J. Conflict resolution that reaps great rewards. *The Journal for Quality and Participation*(23:3), 2000, pp 27-30

[17] Lloyd SR. Conflict resolution: steering clear of the drama triangle. *Rural Telecommunications*(20:5), 2001, pp 30-34

[18] Barnett E. Managing conflicts in systems development. *Hospital Materiel Management Quarterly*(18:4), 1997, pp 1-6

[19] Kerzner H. Conflicts. In: Project Management: A Systems Approach to Planning, Scheduling, and Controlling. New York, NY:John Wiley & Sons, 2001, pp 379-408

[20] Friedman RA, Tidd ST, Currall SC, Tsai JC. What goes around comes around: the impact of personal conflict style on work. *International Journal of Conflict Management*(11:1), 2000, pp 32-55

[21] Rahim MA, Manger NR, Shapiro DL. Do justice perceptions influence styles of handling conflict with supervisors?: what justice perceptions, precisely. *International Journal of Conflict Management*(11:1), 2000, pp 9-31

[22] Barki H, Hartwick J. Interpersonal conflict and its management in information system development. *MIS Quarterly*(25:2), 2001, pp 195-225

[23] Al-Tabtabai H, Alex AP, Aboualfotouh A. Conflict resolution using cognitive analysis approach. *Project Management Journal*(32:2), 2001, pp 4-16

[24] http://en.wikipedia.org/wiki/Project_manager

[25] Project Management: Strategic Design & Implementation, 5th Ed., (2006) David I. Cleland and Lewis R. Ireland

[26] *Project Manager's Portable Handbook*, 2nd Ed. (2004) David I. Cleland and Lewis R. Ireland

[27] http://pmtips.net/project-control/

[28] http://www.made4u.info/wp9/

[29] http://www.interworks.com/blogs/mmedici/2010/01/29/project-management-how-deal-changes-project.

ISSN 2277-3061

Authors

Dr. Kiran Kumar Reddi has received PhD in Computer Science and Engineering from Acharya Nagarjuna University,Guntur,Andhra Pradesh,India. He is working as Assistant Professor in the department of Computer Science,Krishna University,Machilipatnam, Andhra Pradesh, India. His research interest includes Bioinformatics, Data Mining, Image Processing Software Engineering and Network Security. He is a member of professional societies like IETE and ISTE.

S.V. Achuta Rao is Research *Scholar* under the guidance of Dr. Kiran Kumar Reddi in Krishna University, Machilipatnam, India. He has received M.Tech.(Computer Science and Engineering) from JNTU, Kakinada, India. He has been published and presented more than 15 numbers of Research and technical papers in International and National Conferences. His main research interests are Software Engineering and Software Metrics.