Issues & Factors For Evaluation of Software Quality Models

Ranbireshwar S. Jamwal & Deepshikha Jamwal

Jamwal.grandee@gmail.com, jamwal.shivani@gmail.com
Research Scholar, Research Scholar,
Department of Computer Science & IT,
University of Jammu.

ABSTRACT.

Software metrics and quality models play a pivotal role in measurement of software quality. A quality model links together and defines the various software metrics and measurement techniques that an organization uses which when measured. A number of well-known quality models and software metrics are used to build quality software both in industry and in academia. However, during our research on measuring software quality using design patterns, we faced many issues related to existing software metrics and quality models. In this work software quality factors and some of these issues that should be taken into account in very large information systems will be considered and present our approach to software quality assessment [1, 2].

1 INTRODUCTION

As software becomes more and more pervasive, there has been a growing concern in the academic community and in the public about software quality. This concern arises from the acknowledgment that the main objective of software industries in non-military sectors is to balance price and quality to stay ahead of competitors. In the past 15 years, the software industry has created many new different markets, such as open source software and commerce over the Internet. With these new markets, customers of programs now have very high expectations on quality and use quality as a major drive in choosing programs. Some organizations, such as ISO and IEEE, try to standardize software quality by defining models combining and relating software quality characteristics and sub-characteristics. Meanwhile, researchers propose software metrics as tools to measure programs source code, architecture, and performances. Moreover the process of software quality assessment remains an open issue with many models. In this paper, we state some open issues related to software quality, and extend the classical view of software quality models and software metrics. We propose some possible solutions to the mentioned open issues: Modifying software quality models so that characteristics and sub-characteristics are more meaningful to their users; considering the dynamic behavior of the software during its execution. All these solutions can bring significant improvements in the assessment of the quality of programs. In the first part of this paper, we review software assessment tools, some problems, and solutions. There are a number of requirements that need to be met by a quality model, in order for confidence to be gained that the model correctly captures quality requirements, and correctly reflects how well those requirements have been met. A quality model links together and defines the various software metrics and measurement techniques that an organization uses [3].

It requires the development of techniques for very large-scale data handling, efficient strategies for physical clustering of and access to data on secondary storage, and globally distributed data management. It is also widely accepted that software/hardware development can be ad hoc and evolutionary. As a result, engineering environments may start off with poor quality. As the software development evolves so quality should improve. In this work factors in software quality that should be taken into account when using a very large information system will be considered. We start by identifying the metrics and measurement approaches that can be used. A major example will be the lack of suitable performance metrics, which affect many quality factors such as the cost/benefit factor. Performance metrics for distributed software are tied to the target architecture, and there are as many of these as there are distributed architectures. Portability is another major problem; changing computer architecture usually requires the rewriting of programs or re-adapting to a particular architecture and to a particular software environment. In a book on software development and reality construction, the authors acknowledge the dichotomy—in traditional science—between observer and observation and the view of human needs as being outside of the realm of scientific enquiry. They regret this view and state emphasis's theirs: An important aspect of computer science is that it deals with creating reality: The technical reality of the programs executed on the computer, and the condition for the human reality, which unfolds around the computer in use. Therefore, the conceptual categories “true” and “false” [that computer science] relies on are not sufficient in themselves. We have to go beyond them by finding categories for expressing the felicity of our choices, for distinguishing “more or less suitable” as we proceed in making distinctions and decisions in communicative design processes. This is essential for dealing with quality in software development and use [4].

In other fields of science, such as physics, the discovery of relationships among artifacts follows the scientific method.
of observations, laws, and theories. This method consists in recording observations on some facts, then stating laws generalizing the observations, finally developing a theory able to explain the laws and thus the observations. The scientific method begins with facts and observations. We believe that the field of software quality is so far limited by the lack of concrete and consensual facts on the quality of software. Facts in software engineering, in particular about quality, remain too complex because of the many interactions between software artifacts and of the discreet nature of software.

We must find facts in software engineering pertaining to software quality. We believe that design patterns, in particular their design motifs (solutions), could be such facts. Indeed, design motifs are concrete artifacts in programs and consensual in the quality characteristics they bring [5].

2. QUALITY MODELING

Quality is a multidimensional construct reflected in a quality model, where each parameter in the model defines a quality dimension. Many of the early quality models have followed a hierarchical approach in which a set of factors that affect quality are defined, with little scope for expansion (Boehm et al, 1978). More recent models have been developed that follow a 'Define your own' approach (Fenton, 1991). Although an improvement, difficulties arise when comparing quality across projects, due to their tailored nature. The approach used in this work provides a framework in which global and locally defined quality criteria can be considered, individual quality views can be combined and view conflicts can be handled. Thus, it can be used as a standard of excellence measure for a Product, Process or Resource [6].

Considering that quality comprises implicit and/or explicit attributes, there are a number of views and opinions that need to be considered when defining and measuring quality. These views and opinions are referred to as Essential Views, and are determined by examining an individual's expected use of the product, their experiences in developing or using similar products. By concentrating on removing conflicts of opinion between the Essential Views, a consensus can be reached as to what properties constitute quality, and how quality should be measured.

The properties that constitute the 'explicit and / or implicit attributes' of quality form a set of Key Quality Factors and a set of Locally Defined Factors (Horgan, 2000; Horgan et al, 1999). The Key Quality Factors (KQFs) represent global quality criteria, i.e. factors that are required of all products, and their list of properties is static. The Locally Defined Factors (LDFs) represent local quality criteria, i.e. additional factors identified by the Essential Views, and are appropriate only to the current product being developed. In this way, the KQFs represent a common set of criteria that can be used for cross-project comparisons, whilst the LDFs retain the ability to allow local tailoring (figure 1). The LDFs are not a replacement for the KQFs. Instead, they define additional quality criteria [7]. Their identification and inclusion is entirely the responsibility of the Essential Views. If the different views agree that additional criteria are required, then the additional criteria form the LDFs. In this work we only consider Key Quality Factors and for the rest of the paper they are referred to as simply Quality Factors.

![Figure 1: Global and local quality factor.](image)

2.1 QUALITY FACTORS

Quality factors have been used in literature since the early hierarchical quality models (Boehm et al, 1978). The popularity of these is reflected in the fact that the International Standard ISO 9126 is based on them. The standard recommends a number of factors such as Reliability, usability, maintainability etc (Kitchenham, and Pfleeger, 1996). However, people tend to resist plans, which evaluate many quality factors, due to limited resources or tight schedules. Based on previous research (Miyoshi and Azuma, 1993) the number of key factors should be kept between three and eight.

In this work a total, eight Quality Factors are defined. These are Performance, Scalability, Cost/Benefit, Usability, Portability, Robustness, Correctness, and Reliability. Many quality factors would be applied in similar way for sequential and distributed systems, as mentioned earlier we are mainly concerned with those specifics to distributed systems. Thus, many factors will not be included such as Maintainability, which is defined as the ability of a product to be modified, and Timeliness, which is defined as the ability of a product to meet delivery deadlines.

Parallel/distributed processing offers the possibility of an increase in speed and memory beyond the technological limitations of single-processor systems. The performance of such systems can be varied and complex and users need to be able to understand and correct performance problems. Using distributed techniques it is possible to achieve higher throughput, efficiency, performance, and other advantages.

3 OPEN ISSUES

The following is the result of our idea about modification and improvement of existing tools for software quality assessment.

3.1 Human Estimation

A person can look at source code or software products and judge their performance and their quality. Human evaluation is the best source of measurement of software quality because at the it is a person who will deal with quality of software.
The Problem:
- Different taste, different value: Often, software evaluation of one person cannot be expanded as acceptable evaluation for other people because different people have different view on quality. For example, just listen to other people advising for choosing an operating system or a word processor.
- Assessing the quality of software by your own is not practical: It is impossible that everybody have the knowledge and the ability to evaluating the software performance and quality. In addition it is a very hard and time-consuming task.

The solution: Birds of a feather flock together: We must categorize the people who deal with software at different level by considering their need for software quality, and then we can create tailored models for each group, or range of values which are acceptable for similar people. For example, end users mostly have similar ideas about quality of software, but these ideas maybe different from those of people who deal with the maintenance of the same software [8].

3.2 Software Metrics
Software metrics are defined as “standard of measurement, used to judge the attributes of something being measured, such as quality or complexity, in an objective manner”, but subjective measurement of quality comes from human estimation.

The Problem:
- **Evaluation of software code is not enough**: We believe that considering a source code with no regard for its execution is the same as considering the body without its spirit. Well-known metrics are just computing size, filiations, cohesion, coupling, and complexity. These internal attributes are related to code but the quality of a software does not depend on its code only: Acceptable quality in code evaluation does not guarantee performance and quality of software in execution with respect to the user's expectation.
- **Acceptable value for metrics evaluation**: With different views of quality, it is hard to find a numerical value for quality, which could be acceptable by all the people. Also, having different views affects software categorization in certain classification by considering the numerical value as the only parameter on software evaluation.

The Solution: Code without value of execution is not valuable: The value of software metrics must be modified by runtime values for better results. Also, using a good structure and patterns (such as design patterns) in the software design and resulting architecture could increase the software quality. Thus, we want to consider the architectural quality of software by considering the use of design patterns (or lack thereof) in the software architecture.

3.3 Quality Model
A quality model is a schema to better explain of our view of quality. Some existing quality models can predict fault-proneness with reasonable accuracy in certain contexts. Other quality models attempt at evaluating several quality characteristics but fail at providing reasonable accuracy, from lack of data mainly. We take a less “quantitative” approach than quality models counting, for example, numbers of errors per classes and linking these numbers with internal attributes. We favour a more “qualitative” approach linking quality characteristics related to the maintainers’ perception and work directly [9].

The Problem:
- Are all sub-characteristics equal in affecting software characteristics: In the literature, quality models define the relation between quality characteristics and sub-characteristics. For example: Adaptability and Insatiability are two sub-characteristics related to Portability, the question is: If we assess the value of Adaptability as A and the value of Insatiability as B, then is the value of Portability equals to A+B or 2 3A + 1 3B or . . .
- **Main concepts of quality are missing**: In 384 BCE, Aristotle, as a scientist, knew all about medicine, philosophy .In 2005 AD, the concept of quality is the same as science in the age of Aristotle: Quality does not distribute in specific part, when we talk about software quality, we talk about assessing entire items which are part of the concept of quality.

**The Solution:**
- **Coefficient**: Quality as an objective value is dependent on sets of software attributes and customer's requirements. These attributes are explained as different level of characteristics and sub-characteristics in models of quality, but the relation and impact of each characteristic and sub-characteristic should be distinguished. Models can be made more meaningful for different persons by using coefficients, which relate characteristic and sub-characteristic.

4. Our approach to Software Quality Evaluation
To highlight some solutions of the above-mentioned problems, we deal with the 9 steps needed to apply our approach to software quality evaluation, which solves some of the open issues.

4.1 Step by Step:
The following steps highlight the main ideas to implement software quality assessment while considering human requirements.

**Step1**: Choosing Category of People. We must choose at least a person from the category of people, which our software evaluation will be implemented for, for example: Programmers, End-user.

**Step2**: Identifying Sample Program. We must choose a simple program (BP) to be considered as sample evaluation set of our model.

**Step3**: Building a Quality Model. The process of building a quality model decomposes in two main tasks generally:
- Choosing a super-characteristic.
- Choosing and organizing characteristics related to super-characteristic [10].

In our case study, we consider design patterns especially as bridge between internal attributes of programs, external quality characteristics, and software engineers.

**Step4**: Human Evaluation. The small group, or at least one person from the group, must look in the program or product BP and evaluate the quality characteristics we defined in our
quality model, the evaluation could be in form of numerical value or different levels on a Likert scale.

**Step5: Computing Software Metrics over BP.** By using software metrics we evaluate BP numerical values related to software internal attributes.

**Step6: Machine Learning Tools.** JRip as machine learning algorithm generate the relation between human evaluation of software quality (result from Step4) and value of software metrics (result from Step5). The WEKA’s out put consider as set of RULE (an example is shown in Table1) to be used for other software evaluation.

If \((LCOM5 \cdot 1.1) \cdot (NOA \cdot 33.25)\) then (Learnability = Good) else (Learnability = Fair)

**Step7: Computing Software Metrics over EP.** Software metrics are used to assess the values of internal attributes over the EP in the same way as they were for the evaluation of BP.

**Step8: Adapting Metric.** By using ratio over the values from Step7 and Step5, we can relate the numerical values of Step7 with those of Step5. The following method will be used for relation evaluation:

**Phase1.** Finding the Max and Min value of each metrics in EP.

**Phase2.** Finding the Max and Min value of same metrics we were compute on Phase1 over the BP.

**Phase3.** Analyzing the ratio for the values from Phase1 plus values we have in RULE, we build a new RULE compatible with EP. For example: consider the RULE in Table1, considering that upper range and lower range of metrics NOA in BP is UBP NOA and LBP NOA, then the new RULE for Learnability is presented in Table2.

\[
\text{if } (LCOM5 \leq \frac{(NOA \cdot 33.25) \cdot (11.1 \cdot LCOM5)}{(NOA \cdot 33.25) \cdot (11.1 \cdot LCOM5) + (UPP \cdot LCOM5)})
\]

\[
\text{\quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad } \land \quad (NOA \leq \frac{(UPP \cdot LCOM5)}{(NOA \cdot 33.25) \cdot (11.1 \cdot LCOM5)})
\]

\[
\text{\quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad } \quad \text{then } (\text{Learnability = Good})
\]

\[
\text{\quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad } \quad \text{else } (\text{Learnability = Fair})
\]

**Step9: Software Evaluation.** Now, we can evaluate other programs (EP) by applying the of adjusted RULE (from Step8) and software metrics evaluation over the EP.

5. QUALITY FACTORS

Quality factors have been used in literature since the early hierarchical quality models (Boehm et al, 1978). The standard recommends a number of factors such Reliability, usability, maintainability etc (Kitchenham and Pfleeger, 1996). However, people tend to resist plans, which evaluate many quality factors, due to limited resources or tight schedules. Based on previous research (Miyoshi and Azuma, 1993) the number of key factors should be kept between three and eight [10].

In this work total, eight Quality Factors are defined. These are Performance, Scalability, Cost/Benefit, Usability, Portability, Robustness, Correctness, and Reliability. Many quality factors would be applied in similar way for sequential and distributed systems, as mentioned earlier we are mainly concerned with those specifics to distributed systems. Thus, many factors will not be included such as Maintainability, which is defined as the ability of a product to be modified, and Timeliness, which is defined as the ability of a product to meet delivery deadlines. If a product is delivered late, then it may have good quality aspects, but customers may rightly consider it to be of lesser quality than products delivered on time (Horgan, 2000).

Parallel/distributed processing offers the possibility of an increase in speed and memory beyond the technological limitations of single-processor systems. The performance of such systems can be varied and complex and users need to be able to understand and correct performance problems. Using distributed techniques it is possible to achieve higher throughput, efficiency, performance, and other advantages. Although parallel execution time, concurrency, scalability, and speed-up have been proposed in the literature as performance metrics, the uses of speed-up dominate the literature. The notion of speed-up was initially used to estimate the performance benefit of a multiprocessor over a uniprocessor (Kuck, 1978). Later, speed-up has been proposed as a metric to estimate the performance of parallel and distributed algorithms (Ghosh and Yu, 1998) and parallel processor architectures (Manwaring et al, 1994). The most common definition of speed-up is the ratio of the execution time of the best-known sequential algorithm to the execution time of the parallel algorithm on a given number of concurrent processors. The execution time corresponding to a parallel execution is the longest of the CPU times required by any of the concurrent processors [11].

A general, commonsense definition of performance is presented in (Ferrari, 1978) wherein he defines performance as an indication of how well a system, already assumed to be correct, works. In general, the performance metric for distributed systems must reflect the program, the architecture, and the implementation strategies for it depend on each one of them. However, while the parallel execution time, speed-up, and efficiency serve as well known performance metrics, the key issue is the identification of the distributing processing overheads which sets a limit on the speed-up for a given architecture, problem size, and algorithm.

Every problem contains an upper bound on the number of processors, which can be meaningfully employed in its solution. Additional processors beyond this number will not improve solution time and can indeed be detrimental. This upper bound provides an idea as to how suitable a problem is for parallel implementation: a measure of its scalability. For scalability, we look for the ability to handle large numbers of processes and large or long-running programs on increasing number of processors.

Cost / Benefit is defined as the ability of a product to satisfy its cost/benefit specification. The Costs and Benefits involved in a product’s creation should be a major
consideration (Turnball, 1991). If the costs are high, and the benefits of its development are low, then there is little point in developing the product.

Usability is defined as the ability of a product to be used for the purpose chosen. It is a factor that is also considered important in other models (Gillies, 1992; Dromey, 1995). If a product isn't usable, then there is little point in its existence. To be useful, a tool should have adequate documentation and support, and should have an intuitive easy-to-use interface. On-line help is also helpful for usability. Adequate functionality should be provided to accomplish the intended task without putting undue burden on the user to carry out low-level tasks.

Because of the short lifespan of high performance computing platforms and because many applications are developed and run in a distributed heterogeneous environment, most parallel programmers will work on a number of platforms simultaneously or over time. Programmers are understandably reluctant to learn a new performance tool every time they move to a new platform. Thus, we consider portability to be an important feature.

For robustness, we expected the product to crash infrequently and its features to work correctly. Research tools are not expected to be as robust as commercial tools, but if the tool has been released for public use, considerable effort should still have been invested in debugging it and on error handling. Correctness is defined as the ability of a product to meet and support its functional objectives. Other models also include this factor (Fenton, 1991). If software doesn't meet its objectives, then it may be reliable and it may be delivered on time, but no one will use it. Reliability is defined as the ability of a product to reproduce its function over a period of time, and is also included in other approaches (Kitchenham, 1987).

Based on previous research, the number of key factors should be kept between three and eight (Miyoshi and Azuma, 1993). The Quality Factors set were chosen for their obvious importance for the particular system. However, it is accepted that only empirical validation across a large number of projects can determine the completeness of this set [12].

5.1 Relationship chart
The first step of the conflict removal mechanism is implemented by use of a Relationship Chart (Gillies, 1992). The chart displays graphically the relationships between quality criteria as a first stage towards measuring the criteria, and provides the basis for constraints on what can be achieved. In the Relationship Chart, each criterion is listed horizontally and vertically. Where one criterion crosses another, the relationship between those criteria is specified. The relationships for the Quality Factors are fixed. Figure 2 shows the Relationship Chart for the discussed earlier Quality Factors.

By considering these relationships, checks can be made as to the feasibility of requirements. For example, users may state that a reliable product is required, that is both scalable and portable. The relationships between Reliability and Portability, and Portability and Scalability are set to Neutral. Therefore, it is acceptable to state a requirement for a reliable product that is also portable. Similarly, the relationship between Portability and Usability is set to Direct so it is also acceptable to state that a product be portable and usable. As a result, it is an acceptable requirement for a product to be reliable, usable and portable. Note that the Relationship Chart is only a tool and it is still necessary to check the detail of what is being asked [13].

5.2 Polarity profile
The second step in producing a consensus view of quality is to set the required goals for each criterion, based on the relationships identified in the Relationship Chart. In other approaches, a pie chart is used to represent quality goals (Pfleeger, 1993). There is a need to ensure that anyone can understand the graphical format chosen quickly and easily, particularly when it is considered that some essential views may belong to individuals with little technical background [14].

CONCLUSIONS
In the approach presented in this paper individual quality views can be combined, view conflicts can be removed. It allows the specification of benchmarks against which achieved quality levels can be evaluated, and provides guidance for building quality into software for parallel systems. The feasibility of quality goals is controlled by the use of a Relationship Chart and a Polarity Profile. Currently, a set of formal guidelines has yet to be finalized for identifying the Essential Views, despite their importance to the approach. For each occasion that the approach is used, time is required to identify the Essential Views and for those views to derive a consensus of opinions. The solution chosen, therefore, is to use a Polarity Profile. For each criterion, a range of values exists. The Required Quality of a criterion is defined as a single value on a horizontal line. Further, it is easy to determine whether or not a criterion has been over-engineered, since its Actual Quality value will be further advanced along the line than its Required Quality value. The criteria listed in the Polarity Profile are the same criteria as listed in the Relationship Chart.

Each organization will use different metrics and metric approaches to measure different quality attributes [15]. These metrics may be similar, identical or entirely different to those used by other organizations. In order to identify the
Required Quality for each criterion in the Polarity Profile, the expected properties of that criterion need to be expressed using metrics. The same metrics should be used to identify the Actual Quality for that criterion. There is a need, therefore, for Conversion Mechanisms, which convert the results of metrics used to measure the quality of a criterion. However, for each criterion, the Conversion Mechanism will probably be unique to each metric used. Since different organizations may use different metrics, no single Conversion Mechanism will be suitable in all cases. The Conversion Mechanisms used, therefore, should be agreed between the Essential Views.

To our best knowledge, the method we presented is new but still we were using the classical tools of software engineering.

FUTURE SCOPE:
- We have been further working on the analysis of software quality factors and issues particularly in software companies in India. The authors are trying to find the more reliable issues which should be resolved to provide more software quality to different organizations.
- Particularly we are trying to solve the quality related issues and factors which effect the software development in J&K state.

REFERENCES