Software quality

6.1 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;[1]
- 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.

Structural quality is evaluated through the analysis of the software inner structure, its source code, at the unit level, the technology level and the system level, which is in effect how its architecture adheres to sound principles of software architecture outlined in a paper on the topic by OMG.[2] In contrast, functional quality is typically enforced and measured through software testing.

Historically, the structure, classification and terminology of attributes and metrics applicable to software quality management have been derived or extracted from the ISO 9126-3 and the subsequent ISO 25000:2005[3] quality model, also known as SQuaRE.[*citation needed*] Based on these models, the Consortium for IT Software Quality (CISQ) has defined five major desirable structural characteristics needed for a piece of software to provide business value: Reliability, Efficiency, Security, Maintainability and (adequate) Size.

Software quality measurement quantifies to what extent a software or system rates along each of these five dimensions. An aggregated measure of software quality can be computed through a qualitative or a quantitative scoring scheme or a mix of both and then a weighting system reflecting the priorities. This view of software quality being positioned on a linear continuum is supplemented by the analysis of "critical programming errors" that under specific circumstances can lead to catastrophic outages or performance degradations that make a given system unsuitable for use regardless of rating based on aggregated measurements. Such programming errors found at the system level represent up to 90% of production issues, whilst at the unit-level, even if far more numerous, programming errors account for less than 10% of production issues. As a consequence, code quality without the context of the whole system, as W. Edwards Deming described it, has limited value.

To view, explore, analyze, and communicate software quality measurements, concepts and techniques of information visualization provide visual, interactive means useful, in particular, if several software quality measures have to be related to each other or to components of a software or system. For example, software maps represent a specialized approach that "can express and combine information about software development, software quality, and system dynamics".[4]

6.2 Motivation

"A science is as mature as its measurement tools," (Louis Pasteur in Ebert Dumke, p. 91). Measuring software quality is motivated by at least two reasons:

- Risk Management: Software failure has caused more than inconvenience. Software errors have caused human fatalities. The causes have ranged from poorly designed user interfaces to direct programming errors. An example of a programming error that led to multiple deaths is discussed in Dr. Leveson's paper.[5] This resulted in requirements for the development of some types of software, particularly and historically for software embedded in medical and other devices that regulate critical infrastructures: "[Engineers who write embedded software] see Java programs stalling for one third of a second to perform garbage collection and update the user interface, and they envision airplanes falling out of the sky.".[6] In the United States, within the Federal Aviation Administration (FAA), the FAA Aircraft Certification Service provides software programs, policy, guidance and training, focus on software and Complex Electronic Hardware that has an effect on the airborne product (a "product" is an aircraft, an engine, or a propeller).[7]
- Cost Management: As in any other fields of engineering, an application with good structural software quality costs less to maintain and is easier to understand and change in response to pressing business needs. Industry data demonstrate that poor application structural quality in core business applications (such as Enterprise Resource Planning (ERP), Customer Relationship Management (CRM) or large transaction processing systems in financial services) results in cost and schedule overruns and creates waste in the form of rework (up to 45% of development time in some organizations [8]). Moreover, poor structural quality is strongly correlated with high-impact

business disruptions due to corrupted data, application outages, security breaches, and performance problems.

However, the distinction between measuring and improving software quality in an embedded system (with emphasis on risk management) and software quality in business software (with emphasis on cost and maintainability management) is becoming somewhat irrelevant. Embedded systems now often include a user interface and their designers are as much concerned with issues affecting usability and user productivity as their counterparts who focus on business applications. The latter are in turn looking at ERP or CRM system as a corporate nervous system whose uptime and performance are vital to the well-being of the enterprise. This convergence is most visible in mobile computing: a user who accesses an ERP application on their smartphone is depending on the quality of software across all types of software layers.

Both types of software now use multi-layered technology stacks and complex architecture so software quality analysis and measurement have to be managed in a comprehensive and consistent manner, decoupled from the software's ultimate purpose or use. In both cases, engineers and management need to be able to make rational decisions based on measurement and fact-based analysis in adherence to the precept *"In God (we) trust. All others bring data"*. ((mis-)attributed to W. Edwards Deming and others).

CISQ's quality model

Even though "quality is a perceptual, conditional and somewhat subjective attribute and may be understood differently by different people" (as noted in the article on quality in business), software structural quality characteristics have been clearly defined by the Consortium for IT Software Quality (CISQ). Under the guidance of Bill Curtis, co-author of the Capability Maturity Model framework and CISQ's first Director; and Capers Jones, CISQ's Distinguished Advisor, CISQ has defined five major desirable characteristics of a piece of software needed to provide business value.[18] In the House of Quality model, these are "Whats" that need to be achieved:

- Reliability: An attribute of resiliency and structural solidity. Reliability measures the level of risk and the likelihood of potential application failures. It also measures the defects injected due to modifications made to the software (its “stability” as termed by ISO). The goal for checking and monitoring Reliability is to reduce and prevent application downtime, application outages and errors that directly affect users, and enhance the image of IT and its impact on a company's business performance.

- Efficiency: The source code and software architecture attributes are the elements that ensure high performance once the application is in run-time mode. Efficiency is especially important for applications in high execution speed environments such as algorithmic or transactional processing where performance and scalability are paramount. An analysis of source code efficiency and scalability provides a clear picture of the latent business risks and the harm they can cause to customer satisfaction due to response-time degradation.
- Security: A measure of the likelihood of potential security breaches due to poor coding practices and architecture. This quantifies the risk of encountering critical vulnerabilities that damage the business.
- Maintainability: Maintainability includes the notion of adaptability, portability and transferability (from one development team to another). Measuring and monitoring maintainability is a must for mission-critical applications where change is driven by tight time-to-market schedules and where it is important for IT to remain responsive to business-driven changes. It is also essential to keep maintenance costs under control.
- Size: While not a quality attribute per se, the sizing of source code is a software characteristic that obviously impacts maintainability. Combined with the above quality characteristics, software size can be used to assess the amount of work produced and to be done by teams, as well as their productivity through correlation with time-sheet data, and other SDLC-related metrics.

Software functional quality is defined as conformance to explicitly stated functional requirements, identified for example using Voice of the Customer analysis (part of the Design for Six Sigma toolkit and/or documented through use cases) and the level of satisfaction experienced by end-users. The latter is referred as to as usability and is concerned with how intuitive and responsive the user interface is, how easily simple and complex operations can be performed, and how useful error messages are. Typically, software testing practices and tools ensure that a piece of software behaves in compliance with the original design, planned user experience and desired testability, i.e. a piece of software's disposition to support acceptance criteria.

The dual structural/functional dimension of software quality is consistent with the model proposed in Steve McConnell's Code Complete which divides software characteristics into two pieces: internal and external quality characteristics.

External quality characteristics are those parts of a product that face its users, where internal quality characteristics are those that do not.[19]

Alternative approaches

One of the challenges in defining quality is that "everyone feels they understand it"[20] and other definitions of software quality could be based on extending the various descriptions of the concept of quality used in business.

Dr. Tom DeMarco has proposed that "a product's quality is a function of how much it changes the world for the better."[21] This can be interpreted as meaning that functional quality and user satisfaction are more important than structural quality in determining software quality.

Another definition, coined by Gerald Weinberg in Quality Software Management: Systems Thinking, is "Quality is value to some person." [22][23] This definition stresses that quality is inherently subjective—different people will experience the quality of the same software differently. One strength of this definition is the questions it invites software teams to consider, such as "Who are the people we want to value our software?" and "What will be valuable to them?".

Measurement

Although the concepts presented in this section are applicable to both structural and functional software quality, measurement of the latter is essentially performed through testing [see main article: Software Testing].

Introduction

Relationship between software desirable characteristics (right) and measurable attributes (left).

Software quality measurement is about quantifying to what extent a system or software possesses desirable characteristics. This can be performed through qualitative or quantitative means or a mix of both. In both cases, for each desirable characteristic, there are a set of measurable attributes the existence of which in a piece of software or system tend to be correlated and associated with this characteristic. For example, an attribute associated with portability is the number of target-dependent statements in a program. More precisely, using the Quality Function Deployment approach, these measurable attributes are the "hows" that need to be enforced to enable the "whats" in the Software Quality definition above.

The structure, classification and terminology of attributes and metrics applicable to software quality management have been derived or extracted from the ISO 9126-3 and the subsequent ISO/IEC 25000:2005 quality model. The main focus is on internal structural quality. Subcategories have been created to handle specific areas like business application architecture and technical characteristics such as data access and manipulation or the notion of transactions.

The dependence tree between software quality characteristics and their measurable attributes is represented in the diagram on the right, where each of the 5 characteristics that matter for the user (right) or owner of the business system depends on measurable attributes (left):

- Application Architecture Practices
- Coding Practices
- Application Complexity
- Documentation
- Portability
- Technical & Functional Volume

One of the founding member of the Consortium for IT Software Quality, the OMG (Object Management Group), has published an article on "How to Deliver Resilient, Secure, Efficient, and Easily Changed IT Systems in Line with CISQ Recommendations" that states that correlations between programming errors and production defects unveil that basic code errors account for 92% of the total errors in the source code. These numerous code-level issues eventually count for only 10% of the defects in production. Bad software engineering practices at the architecture levels account for only 8% of total defects, but consume over half the effort spent on fixing problems, and lead to 90% of the serious reliability, security, and efficiency issues in production.[24]

Code-based analysis

Many of the existing software measures count structural elements of the application that result from parsing the source code for such individual instructions (Park, 1992),[25] tokens (Halstead, 1977),[26] control structures (McCabe, 1976), and objects (Chidamber & Kemerer, 1994).[27]

Software quality measurement is about quantifying to what extent a system or software rates along these dimensions. The analysis can be performed using a qualitative or quantitative approach or a mix of both to provide an aggregate view [using for example weighted average(s) that reflect relative importance between the factors being measured].

This view of software quality on a linear continuum has to be supplemented by the identification of discrete Critical Programming Errors. These vulnerabilities may not fail a test case, but they are the result of bad practices that under specific circumstances can lead to catastrophic outages, performance degradations, security breaches, corrupted data, and myriad other problems (Nygard, 2007)[28] that make a given system de facto unsuitable for use regardless of its rating based on aggregated measurements. A well-known example of vulnerability is the Common Weakness Enumeration (Martin, 2001),[29] a repository of vulnerabilities in the source code that make applications exposed to security breaches.

The measurement of critical application characteristics involves measuring structural attributes of the application's architecture, coding, and in-line documentation, as displayed in the picture above. Thus, each characteristic is affected by attributes at numerous levels of abstraction in the application and all of which must be included calculating the characteristic's measure if it is to be a valuable predictor of quality outcomes that affect the business. The layered approach to calculating characteristic measures displayed in the figure above was first proposed by Boehm and his colleagues at TRW (Boehm, 1978)[30] and is the approach taken in the ISO 9126 and 25000 series standards. These attributes can be measured from the parsed results of a static analysis of the application source code. Even dynamic characteristics of applications such as reliability and performance efficiency have their causal roots in the static structure of the application.

Structural quality analysis and measurement is performed through the analysis of the source code, the architecture, software framework, database schema in relationship to principles and standards that together define the conceptual and logical architecture of a system. This is distinct from the basic, local, component-level code analysis typically performed by development tools which are mostly concerned with implementation considerations and are crucial during debugging and testing activities.

Reliability

The root causes of poor reliability are found in a combination of non-compliance with good architectural and coding practices. This non-compliance can be detected by measuring the static quality attributes of an application. Assessing the static attributes underlying an application's reliability provides an estimate of the level of business risk and the likelihood of potential application failures and defects the application will experience when placed in operation.

Assessing reliability requires checks of at least the following software engineering best practices and technical attributes:

- Application Architecture Practices
- Coding Practices
- Complexity of algorithms
- Complexity of programming practices
- Compliance with Object-Oriented and Structured Programming best practices (when applicable)
- Component or pattern re-use ratio
- Dirty programming
- Error & Exception handling (for all layers - GUI, Logic & Data)
- Multi-layer design compliance
- Resource bounds management
- Software avoids patterns that will lead to unexpected behaviors
- Software manages data integrity and consistency
- Transaction complexity level

Depending on the application architecture and the third-party components used (such as external libraries or frameworks), custom checks should be defined along the lines drawn by the above list of best practices to ensure a better assessment of the reliability of the delivered software.

Efficiency

As with Reliability, the causes of performance inefficiency are often found in violations of good architectural and coding practice which can be detected by measuring the static quality attributes of an application. These static attributes predict potential operational performance bottlenecks and future scalability problems, especially for applications requiring high execution speed for handling complex algorithms or huge volumes of data.

Assessing performance efficiency requires checking at least the following software engineering best practices and technical attributes:

- Application Architecture Practices
- Appropriate interactions with expensive and/or remote resources

- Data access performance and data management
- Memory, network and disk space management
- Coding Practices
- Compliance with Object-Oriented and Structured Programming best practices (as appropriate)
- Compliance with SQL programming best practices

Maintainability

- Maintainability includes concepts of modularity, understandability, changeability, testability, reusability, and transferability from one development team to another. These do not take the form of critical issues at the code level. Rather, poor maintainability is typically the result of thousands of minor violations with best practices in documentation, complexity avoidance strategy, and basic programming practices that make the difference between clean and easy-to-read code vs. unorganized and difficult-to-read code.[33]

Maintainability is closely related to Ward Cunningham's concept of technical debt, which is an expression of the costs resulting of a lack of maintainability. Reasons for why maintainability is low can be classified as reckless vs. prudent and deliberate vs. inadvertent,[34] and often have their origin in developers' inability, lack of time and goals, their carelessness and discrepancies in the creation cost of and benefits from documentation and, in particular, maintainable source code.[35]

Size

Measuring software size requires that the whole source code be correctly gathered, including database structure scripts, data manipulation source code, component headers, configuration files etc. There are essentially two types of software sizes to be measured, the technical size (footprint) and the functional size:

- There are several software technical sizing methods that have been widely described. The most common technical sizing method is number of Lines Of Code (#LOC) per technology, number of files, functions, classes, tables, etc., from which backfiring Function Points can be computed;
- The most common for measuring functional size is Function Point Analysis. Function Point Analysis measures the size of the software deliverable from a user's perspective. Function Point sizing is done based on user requirements

> and provides an accurate representation of both size for the developer/estimator and value (functionality to be delivered) and reflects the business functionality being delivered to the customer. The method includes the identification and weighting of user recognizable inputs, outputs and data stores. The size value is then available for use in conjunction with numerous measures to quantify and to evaluate software delivery and performance (Development Cost per Function Point; Delivered Defects per Function Point; Function Points per Staff Month.).

The Function Point Analysis sizing standard is supported by the International Function Point Users Group (IFPUG). It can be applied early in the software development life-cycle and it is not dependent on lines of code like the somewhat inaccurate Backfiring method. The method is technology agnostic and can be used for comparative analysis across organizations and across industries.

Since the inception of Function Point Analysis, several variations have evolved and the family of functional sizing techniques has broadened to include such sizing measures as COSMIC, NESMA, Use Case Points, FP Lite, Early and Quick FPs, and most recently Story Points. However, Function Points has a history of statistical accuracy, and has been used as a common unit of work measurement in numerous application development management (ADM) or outsourcing engagements, serving as the "currency" by which services are delivered and performance is measured.

One common limitation to the Function Point methodology is that it is a manual process and therefore it can be labor-intensive and costly in large scale initiatives such as application development or outsourcing engagements. This negative aspect of applying the methodology may be what motivated industry IT leaders to form the Consortium for IT Software Quality focused on introducing a computable metrics standard for automating the measuring of software size while the IFPUG keep promoting a manual approach as most of its activity rely on FP counters certifications.

CISQ announced the availability of its first metric standard, Automated Function Points,to the CISQ membership, in CISQ Technical. These recommendations have been developed in OMG's Request for Comment format and submitted to OMG's process for standardization.

Identifying critical programming errors

Critical Programming Errors are specific architectural and/or coding bad practices that result in the highest, immediate or long term, business disruption risk.

These are quite often technology-related and depend heavily on the context, business objectives and risks. Some may consider respect for naming conventions while others – those preparing the ground for a knowledge transfer for example – will consider it as absolutely critical.

Critical Programming Errors can also be classified per CISQ Characteristics. Basic example below:

- Reliability
    - Avoid software patterns that will lead to unexpected behavior (Uninitialized variable, null pointers, etc.)
    - Methods, procedures and functions doing Insert, Update, Delete, Create Table or Select must include error management
    - Multi-thread functions should be made thread safe, for instance servlets or struts action classes must not have instance/non-final static fields
- Efficiency
    - Ensure centralization of client requests (incoming and data) to reduce network traffic
    - Avoid SQL queries that don't use an index against large tables in a loop
- Security
    - Avoid fields in servlet classes that are not final static
    - Avoid data access without including error management
    - Check control return codes and implement error handling mechanisms
    - Ensure input validation to avoid cross-site scripting flaws or SQL injections flaws
- Maintainability
    - Deep inheritance trees and nesting should be avoided to improve comprehensibility
    - Modules should be loosely coupled (fanout, intermediaries,) to avoid propagation of modifications
    - Enforce homogeneous naming conventions