CHAPTER

Quality concerns in large-scale and complex software-intensive systems

1

Bedir Tekinerdogan[1], Nour Ali[2], John Grundy[3], Ivan Mistrik[4] and Richard Soley[5]

[1]Wageningen University, Wageningen, The Netherlands [2]University of Brighton, Brighton, UK
[3]Swinburne University of Technology, Hawthorn, VIC, Australia [4]Heidelberg, Germany
[5]Object Management Group, Needham, MA, USA

1.1 INTRODUCTION

Since the days of ENIAC (the first computer), computer system developers and their end users have been concerned with quality issues of the resultant systems. Quality comes in many guises and it manifests in many ways. To some, quality relates to the system itself, for example, can it be understood, maintained, extended, scaled, or enhanced? For others, the process of producing the system is their focus, for example, can it be delivered on time, to budget, does it follow best practices and/or relevant standards, and is the process used to develop the system itself of a suitable quality and appropriate for required software quality achievement? Finally, customers, stakeholders, end users and the development team themselves are all concerned, in different ways, whether the system meets its requirements, whether it has been sufficiently verified and/or validated, and does—and can keep on doing—what it was intended to do. A lack of software quality is almost always seen to be highly problematic, again from diverse perspectives.

Nowadays, systems have become very software-intensive, heterogeneous, and very dynamic, in terms of their components, deployment, users, and ultimately their requirements and architectures. Many systems require a variety of mobile interfaces. Many leverage diverse, third-party components or services. Increasingly, systems are deployed on distributed, cloud-based platforms, some diversely situated and interconnected. Multi-tenant systems require supporting diverse users whose requirements may vary, and even change, during use. Adaptive systems need to incorporate various deployment environment changes, potentially including changes in diverse third-party systems. Development processes such as agile methods, outsourcing, and global software development add further complexity and change to software-intensive systems engineering practices.

Software Quality Assurance. DOI: http://dx.doi.org/10.1016/B978-0-12-802301-3.00001-6
© 2016 Elsevier Inc. All rights reserved.

Increasingly, software applications are now "systems of systems" incorporating diverse hardware, networks, software services, and users.

In order to achieve these demanding levels of software quality, organizations and teams need to define and implement a rigorous software quality process. A key to this is defining, for the project at hand, what are the software quality attributes that allow the team, organization, and stakeholders to define quality and required quality levels that must be achieved and maintained. From these attributes, a set of quality requirements for the target system can be defined. Some relate to the functional and non-functional characteristics of the system. Some relate to its static properties, for example, its code, design. Other its run-time properties, for example, behavior, performance, security. Overall system quality must be achieved not only at delivery, but during operation and as the system—and its environment—evolve over time. Quality must be assessed to determine proactively when system quality attributes may fall under a desired threshold and thus quality requirements fail to be met. Mitigations must be applied to ensure these quality requirements are maintained.

Many different kinds of quality challenges present when engineering such systems. Development processes need to incorporate appropriate quality assurance techniques and tools. This includes quality assessment of requirements, architecture, design and target technologies, code bases, and deployment and run-time environments. Software testing has traditionally been a mainstay of such quality assurance, though many other quality management practices are also needed. Testing has become much more challenging with newer development processes, including agile methods, and more complicated, inter-woven service architectures. Because today's complex software-intensive systems are almost invariably composed of many parts, many being third-party applications running on third-party platforms, testing is much more difficult. Adaptive systems that enable run-time change to the software (and sometimes platform) are even more challenging to test, measure quality attributes, and ensure appropriate quality attributes continue to be met. Multiple tenants of cloud applications may each have different requirements—and different views of what "quality" is and how it should be measured and evaluated. Different development teams collaborating explicitly on global software engineering projects—and implicitly on mash-up based, run-time composed systems—may each have differing quality assurance practices, development processes, architectures, technologies, testing tools, and maintenance practices.

A very challenging area of software quality assurance (SQA) is security and privacy. Software-intensive, cloud-hosted, large-scale distributed systems are inherently more vulnerable to attack, data loss, and other problems. Security breaches are one area where—even if all other quality concerns with a software system are met—massively damaging issues can result from a single, severe security problem.

Some software-intensive systems are manifestly requiring of very high levels of quality assurance in software, process, verification and validation, and ongoing maintenance and evolution. Safety-critical systems such as transport (air, rail,

in-vehicle), health, utility (power, gas, water), and financial systems all require very high degrees of holistic SQA practices. These must work in cohesion to ensure a suitable level of quality is able to be achieved at all times.

In this chapter we provide an overview of the SQA domain, with a view to how the advent of software-intensive, large-scale, distributed, complex, and ultimately adaptive and multi-tenant systems have impacted these concepts and practices. Many quality concerns of course remain the same as ever. In many cases, however, achieving them—measuring, assessing, and even defining them—have become much more challenging to software engineers.

The chapter is organized as follows. In Section 1.2 we provide a general discussion on software quality management (SQM) and define the context for SQA. Section 1.3 presents the basic concepts related to software quality models and provides a conceptual model that defines the relation among the different concepts. Section 1.4 discusses the approaches for addressing software quality. Section 1.5 elaborates on assessing system qualities. Section 1.6 presents the current challenges and future directions regarding SQA. Finally, Section 1.7 concludes the chapter.

1.2 SOFTWARE QUALITY MANAGEMENT

Early after the introduction of the first computers and programming languages software became a critical for many organizations. The term "software crisis" was coined at the first NATO Software Engineering Conference in 1968 at Garmisch, Germany. Typically the crisis manifests in different ways including projects exceeding the estimated costs for development, the late delivery of software, and the low quality of the delivered software. Currently, software continues to be a critical element in most large-scale systems and many companies have to cope with a software crisis. To manage the challenges of software development and to ensure the delivery of high quality software, considerable emphasis in the research community has been directed to provide SQM.

SQM is the collection of all processes that ensure that software products, services, and life cycle process implementations meet organizational software quality objectives and achieve stakeholder satisfaction (Galin, 2004; Schulmeyer, 2007; Tian, 2005). SQM comprises three basic subcategories (Figure 1.1): software quality planning (SQP), software quality assurance (SQA), and software quality control (SQC). Very often, like in the Software Engineering Body of Knowledge (Guide to the Software Engineering Body of Knowledge, 2015), software process improvement (SPI) is also described as a separate sub-category of SQM, although it could be included in any of the first three categories.

SQA is an organizational quality guide independent of a particular project. It includes the set of standards, regulations, best practices and software tools to produce, verify, evaluate and confirm work products during the software development life cycle. SQA is needed for both internal and external purposes

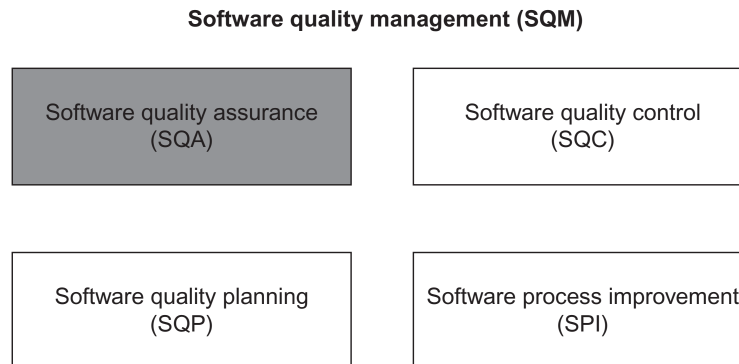

FIGURE 1.1

Context of SQA within the overall SQM process.

(Std. 24765) (ISO/IEC/IEEE 24765:2010(E), 2010). Internal purposes refer to the need for quality assurance within an organization to provide confidence for the management. External purposes of SQA include providing confidence to the customers and other external stakeholders. The IEEE standard (IEEE Std 610.12-1990, 1991) provides the following definitions for *SQA*:

1. a planned and systematic pattern of all actions necessary to provide adequate confidence that an item or product conforms to established technical requirements
2. a set of activities designed to evaluate the process by which products are developed or manufactured
3. the planned and systematic activities implemented within the quality system, and demonstrated as needed, to provide adequate confidence that an entity will fulfill requirements for quality
4. part of quality management focused on providing confidence that quality requirements will be fulfilled.

A SQP is defined at the project level that is aligned with the SQA. It specifies the project commitment to follow the applicable and selected set of standards, regulations, procedures, and tools during the development life cycle. In addition, the SQP defines the quality goals to be achieved, expected risks and risk management, and the estimation of the effort and schedule of software quality activities. A SQP usually includes SQA components as is or customized to the project's needs. Any deviation of an SQP from SQA needs to be justified by the project manager and be confirmed by the company management who is responsible for the SQA.

SQC activities examine project artifacts (e.g., code, design, and documentation) to determine whether they comply with standards established for the project, including functional and non-functional requirements and constraints. SQC ensures thus that artefacts are checked for quality before these are delivered. Example activities of SQC include code inspection, technical reviews, and testing.

SPI activities aim to improve process quality including effectiveness and efficiency with the ultimate goal of improving the overall software quality. In practice, an SPI project typically starts by mapping the organizations' existing processes to a process model that is then used for assessing the existing processes. Based on the results of the assessment an SPI aims to achieve process improvement. In general, the basic assumption for SPI is that a well-defined process will on its turn have a positive impact on the overall quality of the software.

1.3 SOFTWARE QUALITY MODELS

The last decades have shown a growing interest and understanding of the notion of SQA and software quality in general. In this context, a large number of definitions of software quality have emerged. Many of these definitions tend to define quality as conformance to a specification or meeting customer needs. The IEEE ISO/IEC/IEEE 24765 "Systems and software engineering vocabulary" provides the following definition for *quality* (ISO/IEC/IEEE, 2010):

1. the degree to which a system, component, or process meets specified requirements
2. ability of a product, service, system, component, or process to meet customer or user needs, expectations, or requirements
3. the totality of characteristics of an entity that bear on its ability to satisfy stated and implied needs
4. conformity to user expectations, conformity to user requirements, customer satisfaction, reliability, and level of defects present (ISO/IEC 20926:2003)
5. the degree to which a set of inherent characteristics fulfills requirements
6. the degree to which a system, component, or process meets customer or user needs or expectations.

To structure the ideas and provide a comprehensive framework several software quality models have been introduced. A software quality model is a defined set of characteristics, and of relationships between them, which provides a framework for specifying quality requirements and evaluating quality (ISO/IEC 25000:2005) (ISO/IEC, 2011). Usually, software quality models aim to support the specification of quality requirements, to assess existing systems or to predict the quality of a system.

One of the first published quality models is that of McCall (McCall et al., 1977). McCall's model was developed for the US Air Force and is primarily focused on the system developers and the system development process. This model aims to reduce the gap between users and developers by focusing on software quality factors that are important for both users and developers. McCall's quality model adopts three major perspectives for defining software quality: *product revision*, *product transition*, and *product operations*. Product revision relates to the ability to undergo changes, product transition to

the ability to adapt to new environments, and product operations to the operation characteristics of the software. These three types of major perspectives are further decomposed and refined in a hierarchy of 11 quality factors, 23 quality criteria and quality metrics. The main idea of this model is the hierarchical decomposition of quality down to a level at which we can measure and, as such, evaluate quality. In McCall's model, quality factors are defined which describe the external view of the software as defined by the users. Quality factors in turn include quality criteria that describe the internal view of the software as seen by the developer. Finally, for the identified quality criteria the relevant quality metrics are defined to support their measurement and evaluate software quality.

A similar hierarchical model has been presented by Barry W. Boehm (Boehm, 1978) who focuses on *the general utility* of software that is further decomposed into three high-level characteristics including *as-is utility*, *maintainability*, and *portability*. These high-level quality characteristics have in turn seven quality factors that are further decomposed into the metrics hierarchy. Several variations of these models have appeared over time, among which the FURPS that decomposes quality into functionality, usability, reliability, performance and supportability.

The International Organization for Standardization's ISO 9126: Software Product Evaluation: Quality Characteristics and Guidelines for their Use-standard, was inspired by McCall and Boehm models, and also classifies software quality in a structured set of characteristics and sub-characteristics. ISO 9126 has later been revised by ISO/IEC 25010, which now includes ISO25010, and has 8 product quality characteristics and 31 sub-characteristics. The ISO/IEC Standard 9126 and its successor ISO/IEC Standard 25000 (ISO/IEC, 2011) decompose software quality into process quality, product quality, and quality in use.

In the IEEE 24765 Systems and Software Vocabulary the terms software quality factor and software quality attribute are defined as follows:

Software quality factor:

1. A management-oriented attribute of software that contributes to its quality.
2. Higher-level quality attribute.

Software quality attribute:

1. Characteristic of software, or a generic term applying to quality factors, quality sub-factors, or metric values.
2. Feature or characteristic that affects an item's quality.
3. Requirement that specifies the degree of an attribute that affects the quality that the system or software must possess.

To provide a quantitative measure for quality the notion of *metric* is defined:

1. A quantitative measure of the degree to which an item possesses a given quality attribute.

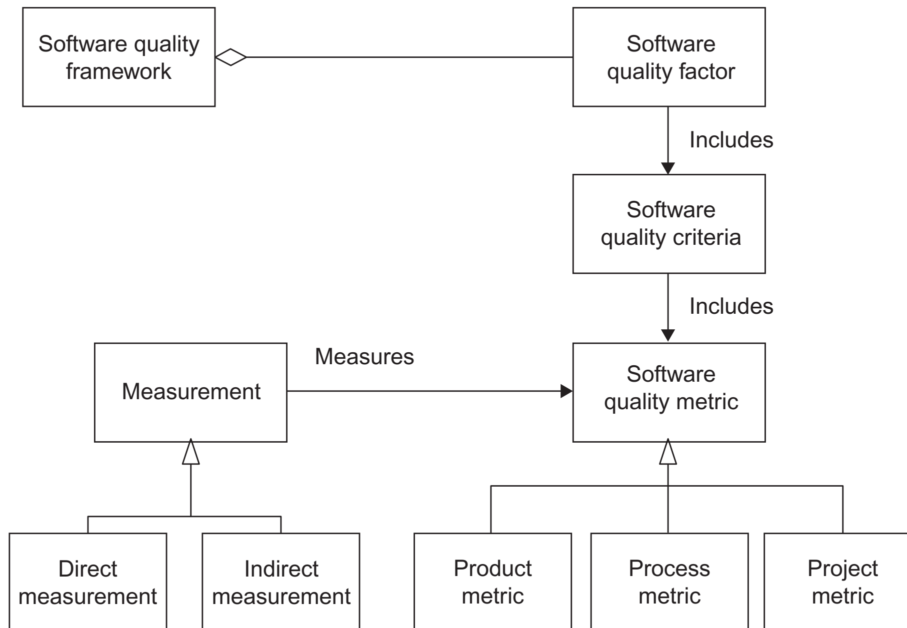

FIGURE 1.2

Conceptual model for SQA.

2. A function whose inputs are software data and whose output is a single numerical value that can be interpreted as the degree to which the software possesses a given quality attribute.

A distinction is made between direct metrics and indirect metrics. A direct metric is "a metric that does not depend upon a measure of any other attribute" (Fenton and Pfleger, 1998). Software metrics are usually classified into three categories: product metrics, process metrics, and project metrics. Product metrics describe the characteristics of the product such as size and complexity. Process metrics describe the characteristics of the software development process. Finally, project metrics describe the project characteristics and execution.

Related to metric is the concept of measurement which is defined as follows:

1. "Measurement is the process by which numbers or symbols are assigned to attributes of entities in the real world in such a way as to characterize them according to clearly defined rules" (Fenton and Pfleger, 1998).
2. "Formally, we define measurement as a mapping from the empirical world to the formal, relational world. Consequently, a measure is the number or symbol assigned to an entity by this mapping in order to characterize an attribute" (Fenton and Pfleger, 1998).

Figure 1.2 shows a conceptual overview of the relations of the above concepts.

1.4 ADDRESSING SYSTEM QUALITIES

SQA can be addressed in several different ways and cover the entire software development process.

Different software development lifecycles have been introduced including waterfall, prototyping, iterative and incremental development, spiral development, rapid application development, and agile development. The traditional waterfall model is a sequential design process in which progress is seen as flowing steadily downwards (like a waterfall) through the phases of Analysis, Design, Implementation, Testing, and Maintenance. The waterfall model implies the transition to a phase only when its preceding phase is reviewed and verified. Typically, the waterfall model places emphasis on proper documentation of artefacts in the life cycle activities. Advocates of agile software development paradigm argue that for any non-trivial project finishing a phase of a software product's life cycle perfectly before moving to the next phases is practically impossible. A related argument is that clients may not know exactly what requirements they need and as such requirements need to be changed constantly.

It is generally acknowledged that a well-defined mature process will support the development of quality products with a substantially reduced number of defects. Some popular examples of process improvement models include the Software Engineering Institute's Capability Maturity Model Integration (CMMI), ISO/IEC 12207, and SPICE (Software Process Improvement and Capability Determination).

Software design patterns are generic solutions to recurring problems. Software quality can be supported by reuse of design patterns that have been proven in the past. Related to design patterns is the concept of anti-patterns, which are a common response to a recurring problem that is usually ineffective and counterproductive. Code smell is any symptom in the source code of a program that possibly indicates a deeper problem. Usually code smells relate to certain structures in the design that indicate violation of fundamental design principles and likewise negatively impact design quality.

An important aspect of SQA is software architecture. Software architecture is a coordination tool among the different phases of software development. It bridges requirements to implementation and allows reasoning about satisfaction of systems' critical requirements (Albert and Tullis, 2013). Quality attributes (Babar et al., 2004) are one kind of non-functional requirement that are critical to systems. The Software Engineering Institute (SEI) defines a quality attribute as "a property of a work product or goods by which its quality will be judged by some stakeholder or stakeholders" (Koschke and Simon, 2003). They are important properties that a system must exhibit, such as scalability, modifiability, or availability (Stoermer et al., 2006).

Architecture designs can be evaluated to ensure the satisfaction of quality attributes. Tvedt Tesoriero et al. (2004), Stoermer et al. (2006) divide architectural evaluation work into two main areas: pre-implementation architecture evaluation,

and implementation-oriented architecture conformance. In their classification, pre-implementation architectural approaches are used by architects during initial design and provisioning stages, before the actual implementation starts. In contrast implementation-oriented architecture conformance approaches assess whether the implemented architecture of the system matches the intended architecture of the system. Architectural conformance assesses whether the implemented architecture is consistent with the proposed architecture's specification, and the goals of the proposed architecture.

To evaluate or design a software architecture at the pre-implementation stage, tactics or architectural styles are used in the architecting or evaluation process. Tactics are design decisions that influence the control of a quality attribute response. Architectural Styles or Patterns describe the structure and interaction between collections of components affecting positively to a set of quality attributes but also negatively to others. Software architecture methods are encountered in the literature to design systems based on their quality attributes such as the Attribute Driven Design (ADD) or to evaluate the satisfaction of quality attributes in a software architectural design such as the Architecture Tradeoff Analysis Method (ATAM). For example, ADD and ATAM follow a recursive process based on quality attributes that a system needs to fulfill. At each stage, tactics and architectural patterns (or styles) are chosen to satisfy some qualities.

Empirical studies have demonstrated that one of the most difficult tasks in software architecture design and evaluation is finding out what architectural patterns/styles satisfy quality attributes because the language used in patterns does not directly indicate the quality attributes. This problem has also been indicated in the literature (Gross and Yu, 2001 and Huang et al., 2006).

Also, guidelines for choosing or finding tactics that satisfy quality attributes have been reported to be an issue in as well as defining, evaluating, and assessing which architectural patterns are suitable to implement the tactics and quality attributes (Albert and Tullis, 2013). Towards solving this issue Bachmann et al. (2003), Babar et al. (2004) describe steps for deriving architectural tactics. These steps include identifying candidate reasoning frameworks which include the mechanisms needed to use sound analytic theories to analyze the behavior of a system with respect to some quality attributes (Bachmann et al., 2005). However, this requires that architects need to be familiar with formal specifications that are specific to quality models. Research tools are being developed to aid architects integrate their reasoning frameworks (Christensen and Hansen, 2010), but still reasoning frameworks have to be implemented, and tactics description and how they are applied has to be indicated by the architect. It has also been reported by Koschke and Simon (2003) that some quality attributes do not have a reasoning framework.

Harrison and Avgeriou have analyzed the impact of architectural patterns on quality attributes, and how patterns interact with tactics (Harrison and Avgeriou, 2007; Harrison and Avgeriou). The documentation of this kind of analysis can aid in creating repositories for tactics and patterns based on quality attributes.

Architecture prototyping is an approach to experiment whether architecture tactics provide desired quality attributes or not, and to observe conflicting qualities (Bardram et al., 2005). This technique can be complementary to traditional architectural design and evaluation methods such as ADD or ATAM (Bardram et al., 2005). However, it has been noted to be quite expensive and that "substantial" effort must be invested to adopt architecture prototyping (Bardram et al., 2005).

Several architectural conformance approaches exist in the literature (Murphy et al., 2001; Ali et al.; Koschke and Simon, 2003). These check whether software conform to the architectural specifications (or models). These approaches can be classified either by using static (source code of system) (Murphy et al., 2001; Ali et al.) or dynamic analysis (running system) (Eixelsberger et al., 1998), or both. Architectural conformance approaches have been explicit in being able to check quality attributes (Stoermer et al., 2006; Eixelsberger et al., 1998) and specifically run-time properties such as performance or security (Huang et al., 2006). Also, several have provided feedback on quality metrics (Koschke, 2000).

1.5 ASSESSING SYSTEM QUALITIES

Sections 1.2–1.4 defined concepts and a plan of how we can realize system quality. In this section, we define some of the metrics relating to system quality and how these are monitored and tracked throughout the software development life cycle. The purpose of using metrics is to reduce subjectivity during monitoring activities and provide quantitative data for analysis, helping to achieve desired software quality levels. In this section we focus on approaches for assessing different quality attributes and suitable metrics relevant to the assessment of these quality attributes.

As discussed above, a huge range of software quality attributes have been identified, ranging from low-level code quality issues to overarching software procurement, development, and deployment processes. Each class of quality attribute has a set of metrics that can be used to assess differing quality dimensions of the software system. Metrics need to be assessed to determine whether the software is meeting—or likely to meet—the required quality thresholds set by stakeholders. The thresholds may vary considerably depending on software size, cost, nature of team, software process being used, software quality framework being used, and so on. With modern, complex software-intensive systems, quality requirements may even vary depending on changes to deployment scenario and end users.

1.5.1 ASSESSMENT PROCESSES

The IEEE Software Quality Metrics Methodology (Huang et al., 2006) is a well-known framework for defining and monitoring system-quality metrics and analysis of measurements gathered through the implementation of metrics. Key goals

of the framework are to provide organizations a standard methodology to assess achievement of quality goals, establish quality requirements for a system, establish acceptance criteria, detect anomalies, predict future quality levels, monitor changes in quality as software is modified, and to help validate a metrics set. A software quality metrics framework is provided to assist achieving these goals.

The first step of the methodology is to establish a set of software quality requirements. This includes identifying possible requirements, determining the requirements to use, and determining a set of metrics to use to measure quality. A set of metrics—an approved metrics set—is then established, including a cost−benefit analysis of implementing and monitoring the metrics and a process of commitment to the established metrics set. The metrics are then implemented on the software project. This includes data collection procedures, a measurement process established, and metric computation from measures. An analysis phase is used to interpret the results from the metrics capture, identifying the levels of software quality being achieved against the requirements targets. Predictions can be made to assist project management and a quality requirements compliance process is implemented to ensure the project is on target. A final step is validating the quality metrics to ensure they provide a suitable set of product and process metrics to predict desired quality levels. A set of validity criteria are used in the assessment of the metrics set and the results are documented and periodically re-validated.

A range of complementary and alternative approaches have been developed to support the software quality assessment process. CMMI (Bardram et al., 2005) includes several components relating to SQM that incorporate aspects of the assessment of quality attributes. In particular, PPQA (product and process quality assurance) and related PMC (project monitoring and control) and MA (measurement and analysis). Higher levels of quality assurance organization include QPM (quantitative project management) and CAR (causal analysis and resolution). Various agile development processes incorporate quality assessment processes. These include several efforts to develop an agile maturity model (AMM) (Patel and Ramachandran, 2009), complementary in many ways to CMMI but incorporating agile concepts of rapid iteration, on-site customer, pair programming and other agile practices, and minimal investment as in spikes and refactoring as and when needed. The move to many cloud-based applications has increased interest in suitable quality assessment processes and techniques for such nontraditional applications where systems are composed from disparate services, many from different providers.

Key issues with any quality assessment processes include:

- Cost vs. benefit of carrying out the assessment—this includes cost to capture suitable measurements, cost to implement, cost to analyze vs. benefit gained in terms of monitoring quality compliance, and predictive quality assessment
- Team adoption and training—including integrating assessment into the development process, ensuring data can be suitably collected and analyzed, and the team can act on problematic quality assessments

- Evolution of both requirements and system—particularly challenging in the context of cloud-based systems and autonomic systems, where new stakeholders and/or deployment environment conditions can dramatically impact overall system quality metrics
- Lack of mitigations if quality requirements are not being met—if the project is failing to meet one or more quality targets, or likely will fail to meet these, suitable actions must be available to address the issue or the project is at risk of failure.

1.5.2 METRICS AND MEASUREMENTS

A set of metrics are required by which quality attributes can be assessed, and metrics have a set of measurements that need to be periodically taken in order to make judgements about the state of product and process quality. Quality assessment requires the following:

- Definition of appropriate metrics/measures to use—how quality attributes will be assessed
- Definition of a set of expected measurement targets—these can be simple thresholds or very complicated calculations based on a number of measurements taken
- A data collection process put in place to periodically take required measurements, including suitable data collection tools identified and deployed on the project
- Data analysis conducted at suitable times and judgements made on quality
- Data storage, reuse and comparison to determine current quality levels, compliance, trends and future predictions
- Data needs to be suitably protected, including data relating to people, financials, and sensitive requirements and/or measurements.

A wide range of models have been developed to specify and capture software quality metrics. We briefly review several here, with a view to newer metrics for cloud-based platforms, agile methods and large-scale software-intensive systems.

The Software Assurance Technology Center (SATC) at NASA introduced a wide range of software quality metrics applicable to most software processes, architectures, programming languages, and testing strategies. These are grouped into several areas. Requirements-level metrics are used to assess the quality of software requirements. This is fundamental as it doesn't really matter how well a team "does the thing right" (i.e., use best practice design, coding, testing, etc.), if they are not "doing the right thing" (i.e., building the right systems to meet stakeholders needs). Requirements-level quality metrics include completeness, correctness, and consistency of the requirements, commonly called the 3Cs (Pohl, 2010). Additional metrics include traceability—the ability to link requirements to design, code, test artefacts, and volatility—how changeable the requirements are and hence impact on system architecture, design, code, deployment, etc.

A great many code-level metrics have been developed. Classic ones include lines of code (often a poor productivity measure), cyclometric complexity, function point analysis, cohesion, coupling, and various kinds of complexity and size analysis. While historically applied to source code, many can also be applied to design-level models, especially when used for model-driven engineering activities, and even potentially to configuration models. A variety of metrics for user interface of systems have also been developed, many derived from HCI and usability research and practice. These have been applied to web interfaces, more recently to mobile interfaces, and are increasingly being applied to ubiquitous, haptic, virtual reality, touch, gesture, speech and other more human-centric interfaces (Albert and Tullis, 2013). Such metrics include simple completion rates (can or can't complete task), task time, user satisfaction, error rates, various interaction measures, and marketing-style metrics like return rates and conversion rates.

Software testing has historically been used as a major quality assurance achievement mechanism in software development. Many metrics have been developed to support quality assurance via testing. These include defects/bugs per lines of code, code coverage of testing (predominantly for unit testing approaches), fault localization, and identification of criticality of located defects (Kan, 2002).

Process-level metrics are used to qualify and quantify quality aspects associated with the software development process employed by a team. Examples include burn-down charts commonly used in agile methods to track progress, task completion rates, critical paths, and hours (and other resources) spent on development and assurance activities (Kitchenham, 1996).

Service-oriented and cloud-based systems have brought new demands to the evaluation of run-time performance of software systems with a view to meeting quality attributes in this area. Such metrics include traditional ones such as service availability, outage duration, mean-time between failures, completion time, and response time for requests (Papazoglou and van den Heuvel, 2003). More recent measures are needed to assess the quality of service and cloud application delivery, including network and storage device capacity, server capacity (in terms of compute power), web server capacity (number of concurrent requests, users supportable, etc.), instance start up/shut down for cloud elasticity measurement, mean-time to switch-over, and mean-time to system recovery after failure (Li et al., 2012).

1.6 CURRENT CHALLENGES AND FUTURE DIRECTIONS OF SOFTWARE QUALITY

A number of major challenges face software teams—and organizations and individual developers and operators—in maintaining software quality for today's, and tomorrow's, software-intensive systems.

Systems seem to grow ever more interdependent, meaning there are few systems that don't depend heavily on other, usually third-party systems, for major aspects of their components and operation and therefore quality. A failure or simply lower-than-acceptable level of quality in any one of these components or connected services may lead to unacceptable quality degradation in the system as a whole. This quality attribute problem may be to do with incorporating unmaintainable or unportable code; insufficient testing of a used service or the service integration; lower than required run-time performance, reliability, or excessive resource utilization; poor usability of integrated interfaces especially on mobile devices; inefficient or ineffective software process used for all or part of the system's development; or a failure in deployment environment or the user community, for example, comprising security of a component and thus the system as a whole. A number of trends increase these problems, some dramatically. The trend to DevOps, or Development-Operations, where the division between developing vs. maintaining a system disappears. The trend to service-oriented architectures and cloud computing platforms where there is huge dependence on others for necessary system infrastructure and indeed critical software components. The use of agile and global software engineering practices puts greater delivery demands and expectations on teams while greatly increasing challenges around team coordination and software management. The adoption of the "internet of things" (IoT) where many system components are software-intensive but rely on very heterogeneous hardware and networking components, themselves prone to various quality challenges.

Future SQA approaches and supporting techniques, tools, and processes will need to address these challenges. Quality processes, measurements, and management must be applied to diverse non-software components of systems, software components, and the system as whole. Run-time evolution of systems including deployment environment, networking, hardware, and integrated services will mean more run-time quality management is necessary. This will need to be paired with software quality meta-practices, that is, software being engineered with greater range of quality attributes measured and managed at run-time as well as development-time. High turn-around of changes in the DevOps paradigm, contracted platform provisioning in the cloud computing paradigm, and diverse integrated data sources in the IoT paradigm, will all need higher degrees and frequency of quality attention than traditional enterprise systems development. Distributed, agile teams and incorporation of large numbers of third-party services all require more precise definition of quality attributes and thresholds, agreement on quality maintaining processes, and more accurate predictive analytics associated with SQA practices.

Finally, big data applications have their own quality challenges, not just around their software systems but data quality, privacy, provenance, and scaling. It is highly likely that future quality assurance techniques and tools themselves need to make use of large-scale data analytics approaches to improve our ability to manage very diverse ranges of quality metrics, size of quality measurements and predictive analysis to proactively tackle emergent software quality problems.

1.7 CONCLUSION

In this chapter we have provided a general overview of software quality concerns and SQM. SQM is the collection of all processes that ensure that software products, services, and life cycle process implementations meet organizational software quality objectives and achieve stakeholder satisfaction. We have briefly described the basic SQM approaches including SQP, SQA, SQC, and SPI. Considering the topic of the book we have focused on SQM and in this context discussed the short history and evolution of software quality models. In addition the current state-of-the-art approaches on assessing system quality approaches have been discussed. Although a large body of knowledge on SQA exists, there are still many great challenges which require attention. The subsequent chapters in this book address some of the identified relevant issues.

REFERENCES

Albert, W., Tullis, T., 2013. Measuring the User Experience: Collecting, Analyzing, and Presenting Usability Metrics. Morgan Kaufmann.

Ali, N., Solis, C., 2014. Exploring how the attribute driven design method is perceived, In: Mistrik, I., Bahsoon, R., Eeles, P., Roshandel, R., Stal, M. (Eds.), Relating System Quality and Software Architecture. Morgan Kaufman Elsevier, United States, pp. 23−40. ISBN 9780124170094.

Ali, N., Rosik, J., Buckley, J. Characterizing real-time reflexion-based architecture recovery: an in-vivo multi-case study. In: Proceedings of the 8th International ACM SIGSOFT Conference on Quality of Software Architectures (QoSA'12). ACM, New York, NY, pp. 23−32.

Babar, M.A., Zhu, L., Jeffery, R., 2004. A framework for classifying and comparing software architecture evaluation methods. In: ASWEC, p. 309.

Bachmann, F., Bass, L., Klein, M., 2003. Deriving Architectural. Tactics: A Step Toward. Methodical Architectural. Design. CMU/SEI-2003-TR-004. ESC-TR-2003-004.

Bachmann, F., Bass, L., Klein, M., Shelton, C., 2005. Designing software architectures to achieve quality attribute requirements. In: Software, IEE Proceedings, vol. 152, issue 4, pp. 153−165.

Bardram, J.E., Christensen, H.B., Corry, A.V., Hansen, K.M., Ingstrup, M., 2005. Exploring quality attributes using architectural prototyping. In: Proceedings of First International Conference on the Quality of Software Architectures, LNCS, vol. 3712, pp. 155−170.

Bass, L., Clements, P., Kazman, R., 2010. Software Architecture in Practice, third ed. Addison-Wesley Professional.

Boehm, B., 1978. Characteristics of Software Quality, Vol 1 of TRW Series on Software Technology. North-Holland, Amsterdam, Holland.

Chrissis, M.B., Konrad, M., Shrum, S., 2003. CMMI Guidlines for Process Integration and Product Improvement. Addison-Wesley Longman Publishing Co., Inc.

Christensen, H.B., Hansen, K.M., 2010. An empirical investigation of architectural prototyping. J. Syst. Softw. 83 (1), 133−142.

Diaz-Pace, A., Kim, H., Bass, L., Bianco, P., Bachmann, F., 2008. Integrating quality-attribute reasoning frameworks in the ArchE design assistant. In: Proceedings of the 4th International Conference on Quality of Software-Architectures: Models and Architectures, LNCS, vol. 5281, pp. 171–188.

Dybå, T., Dingsøyr, T., 2008. Empirical studies of agile software development: a systematic review. Inf. Softw. Technol. 50 (9), 833–859.

Eixelsberger, W., Ogris, M., Gall,H., Bellay, B., 1998. Software architecture recovery of a program family. In: ICSE, pp. 508–511.

Emeakaroha, V.C., et al. 2010. Low level metrics to high level SLAs-LoM2HiS framework: bridging the gap between monitored metrics and SLA parameters in cloud environments. In: 2010 International Conference on High Performance Computing and Simulation (HPCS), IEEE.

Fenton, N.E., Pfleger, S.L., 1998. Software Metrics—A Rigorous and Practical Approach, second ed. International Thomson Press, London.

Franke, D., Weise, C. 2011. Providing a software quality framework for testing of mobile applications. In: 2011 IEEE Fourth International Conference on Software Testing, Verification and Validation (ICST), IEEE.

Galin, D., 2004. Software Quality Assurance: From Theory to Implementation. Pearson Education.

Garlan, D., Schmerl, B., 2004. Using Architectural Models at Runtime: Research Challenges. In: First European Workshop on Software Architecture, LNCS 3047. Springer, pp. 200–205.

Gorton, I., 2006. Essential Software Architecture. Springer-Verlang.

Gross, D., Yu, E., 2001. From non-functional requirements to design through patterns. Requirements Eng. 6 (1), 18–36.

Guide to the Software Engineering Body of Knowledge, 2015. SWEBOK Guide <https://www.computer.org/web/swebok>.

Harrison, N.B., Avgeriou, P., 2007. Leveraging architecture patterns to satisfy quality attributes. In: European Conference on Software Architecture, LNCS, pp. 263–270.

Harrison, N.B., Avgeriou, P., 2010. How do architecture patterns and tactics interact? A model and annotation. J. Syst. Softw. 83 (10), 1735–1758.

Huang, G., Hong, M., Yang, F.Q., 2006. Runtime recovery and manipulation of software architecture of component-based systems. Autom. Softw. Eng. 13 (2), 257–281.

IEEE Std 610.12-1990—IEEE Standard Glossary of Software Engineering Terminology, Corrected Edition, February 1991. In: IEEE Software Engineering Standards Collection, The Institute of Electrical and Electronics Engineers, New York, 1991.

IEEE, 1061-1992—IEEE Standard for a Software Quality Metrics Methodology, IEEE Computer Society, 1992, http://dx.doi.org/10.1109/IEEESTD.1993.115124.

ISO/IEC/IEEE 24765:2010(E)—IEEE Systems and Software Engineering Vocabulary, 2010.

ISO 9000-3:1997(E), Quality Management and Quality Assurance Standards—Part 3: Guidelines for the Application of ISO 9001:1994 to the Development, Supply, Installation and Maintenance of Computer Software, second ed. International Organization for Standardization (ISO), Geneva.

ISO 9000-3:2001 Software and System Engineering—Guidelines for the Application of ISO 9001:2000 to Software, Final draft. International Organization for Standardization (ISO), Geneva, unpublished draft, December 2001.

ISO/IEC Systems and Software Engineering—Systems and Software Quality Requirements and Evaluation (SquaRE)—System and Software Quality Models. ISO/IEC 25010:2011, 2011. Available from: <http://www.iso.org/iso/iso_catalogue/catalogue_tc/catalogue_detail.htm?csnumber=35733>.

Kan, S.H., 2002. Metrics and Models in Software Quality Engineering. Addison-Wesley Longman Publishing Co., Inc.

Kazman, R., Bass, L., Klein, M., 2006. The essential components of software architecture design and analysis. J. Syst. Softw. 79 (8), 1207–1216.

Kitchenham, B.A., 1996. Software Metrics: Measurement for Software Process Improvement. Blackwell Publishers, Inc.

Koschke, R., 2000. Atomic Architectural Component Recovery for Program Understanding and Evolution (Ph.D. thesis). Universität Stuttgart.

Koschke, R., Simon, D., 2003. Hierarchical reflexion models. In: Proceedings of the 10th Working Conference on Reverse Engineering, Victoria, Canada.

Li, Z. et al., 2012. On a catalogue of metrics for evaluating commercial cloud services. In: Proceedings of the 2012 ACM/IEEE 13th International Conference on Grid Computing. IEEE Computer Society.

McCall, J., Richards, P., Walters, G., 1977. Factors in Software Quality, vols. 1–3, NTIS AD-A049-014, 015, 055, November 1977.

Murphy, G., Notkin, D., Sullivan, K., 2001. Software reflexion models: bridging the gap between design and implementation. IEEE Trans. Softw. Eng. 27 (4), 364–380.

Patel, C., Ramachandran, M., 2009. Agile maturity model (AMM): a Software Process Improvement framework for agile software development practices. Int. J. Softw. Eng. 2 (1), 3–28.

Papazoglou, M.P., van den Heuvel, W.J., 2003. Service-oriented computing: state-of-the-art and open research issues. IEEE Comput. 40 (11).

Pohl, K., 2010. Requirements Engineering: Fundamentals, Principles, and Techniques. Springer Publishing.

Remco, C., Van Vliet, H., 2009. QuOnt: an ontology for the reuse of quality criteria. In: ICSE Workshop on Sharing and Reusing Architectural Knowledge, pp. 57–64.

Rozanski, N., Woods, E., 2011. Software Systems Architecture: Working with Stakeholders Using Viewpoints and Perspectives. Addison-Wesley.

Schulmeyer, G., 2007. Handbook of Software Quality Assurance. Artech House Publishers. fourth ed.

Software Engineering Institute, 2010. Software Architecture Glossary. <http://www.sei.cmu.edu/architecture/start/glossary/>.

Stoermer, C., Rowe, A., O'Brien, L., Verhoef, C., 2006. Model-centric software architecture reconstruction. Softw. Pract. Exper. 36 (4), 333–363, ISSN 0038-0644. http://dx.doi.org/10.1002/spe.v36:4.

Tvedt, R.T., Costa, P., Lindvall, M., 2004. Evaluating software architectures. Adv. Comput. 61, 1–43, <http://dblp.uni-trier.de/db/journals/ac/ac61.html#TvedtCL04>.

Tian, J., 2005. Software Quality Engineering: Testing, Quality Assurance, and Quantifiable Improvement. John Wiley & Sons.

Wojcik, R., Bachmann, F., Bass, L., Clements, P., Merson, P., Nord, R., et al., 2006. Attribute-Driven Design (ADD), Version 2.0. Technical Report CMU/SEI-2006-TR-023, SEI.