Chapter 1

SQA – Definitions and Concepts

1.1 Software quality and software quality assurance – definitions

We shall Start by delving into our target topic of software quality and discuss the following basic definitions:

- Software quality
- Software quality assurance (SQA)
- Software quality assurance – an expanded definition
- The objectives of SQA activities

The definition of software quality is shown in Frame 1.1.

Frame 1.1: Software quality – a definition

Source: IEEE Std. 730-2014 (IEEE, 2014)

Software quality is
The degree to which a software product meets established requirements; however, quality depends upon the degree to which established requirements accurately represent stakeholder needs, wants, and expectations.

Two aspects of software quality are presented in the above definition: one is meeting the requirements, while the other is generating customer/stakeholder satisfaction. A high quality software product is expected to meet all written development requirements – whether defined fully before the development began, or later in the course of the development process – and to meet the relevant regulations and professional conventions. Quality is also achieved through fulfillment of stakeholder needs and wants.

Software Quality: Concepts and Practice, First Edition. Daniel Galin.

Software quality assurance – definition

One of the most commonly used definitions of SQA is proposed by the IEEE, cited in Frame 1.2.

Frame 1.2: Software quality assurance – a definition

Source: IEEE Std. 730-2014

Software quality assurance is
A set of activities that define and assess the adequacy of software process to provide evidence that establishes confidence that the software processes are appropriate for and produce software products of suitable quality for their intended processes. A key attribute of SQA is the objectivity of the SQA function with respect to the project. The SQA function may also be organizationally independent of the project, that is, free from technical, managerial, and financial pressures from the project.

This definition may be characterized by the following:

- Plan and implement systematically. SQA is based on the planning and implementation of a series of activities that are integrated into all stages of the software development process. These activities are performed in order to substantiate the client's confidence that the software product will meet all the technical requirements.
- Refer to the software development products keeping the specified technical requirements and suitability for stake holder's intended use. However, it does not include quality of the operation services.
- Refer to the technical appropriateness of the development process. However, important attributes of the development process, namely schedule and budget keeping, are not included. It is noteworthy that:
  - **a.** The appropriateness of project schedule and budget is a major issue in SQA as can be seen by requirement for performing contract reviews and project planning.
  - **b.** The major part of project progress control procedures, given to the issues of schedule and budget.
  - **c.** The close relationships that exist between software product quality, project schedule, and project budget, where schedule and budget failures result, almost always, in unavoidable software quality failure.

An extended SQA definition was created considering the importance of the quality of the software operation and the important effect of schedule and budget keeping on the software quality product.

The resulting expanded SQA definition is shown in Frame 1.3.

Frame 1.3: Software quality assurance – an expanded definition

Software quality assurance
A set of activities that define and assess the adequacy of software process to provide evidence that establishes confidence that the software processes are appropriate for producing software products of suitable quality, for their intended processes, or for their intended operation services and fulfils the requirements of schedule and budget keeping

The objectives of SQA activities

The objectives of SQA activities refer to the functional and managerial aspects of software development and software maintenance. These objectives are listed in Frame 1.4.

Frame 1.4: The objectives of SQA activities

The objectives of SQA activities are
• Ensuring an acceptable level of confidence that the software product and software operation services will conform to functional technical requirements and be suitable quality for its intended use. • According to the extended SQA definition – ensuring an acceptable level of confidence that the software development and software operation process will conform to scheduling and budgetary requirements. • Initiating and managing activities to improve and increase the efficiency of software development, software operation, and SQA activities. These activities yield improvements to the prospects' achieving of functional and managerial requirements while reducing costs.

The other sections of the chapter deal with the following issues:

- What is a software product?
- The principles of SQA
- Software errors, faults, and failures
- The causes of software errors
- Software quality assurance versus software quality control (SQC)
- Software quality engineering and software engineering

1.2 What is a software product?

Intuitively, when we think about software, we imagine an accumulation of programming language instructions and statements, usually referred to as "code."

However, when referring to a professional software product, "code" by itself is not sufficient. Software products need to undergo defect corrections, and other maintenance services, which typically include user instruction, corrections, adaptations, and improvements of the software product during their life cycle. Accordingly, software products also comprise components, required to ensure operational success of the services provided by the product. The ISO/IEC/IEEE definition shown in Frame 1.5 lists these components.

Frame 1.5: Software product definition

Source: ISO/IEC/IEEE Std. 90003:2014 (ISO/IEC/IEEE, 2014)

Software product is
Set of computer programs, procedures, and possibly associated documentation and data.

The software product components are:

Computer programs "the code". The computer programs activate the computer system to perform the required applications. The computer programs include several types of code, such as source code, executable code, test code, and so on.

Procedures. Procedures define the order and schedule within which the software or project programs are performed, the method for handling common malfunctioning of software products, and so on.

Documentation. The purpose of the documentation is to instruct or support new software product version developers, maintenance staff, and end users of the software product. It includes the various design reports, test reports, and user and software manuals, and so on.

Data necessary for operating the software system. The required data include lists of codes and parameters, and also standard test data. The purpose of the standard test data is to ascertain that no undesirable changes in the code or software data have occurred during bug corrections and other software maintenance activities, and to support the detection of causes for any malfunctioning.

To summarize the above discussion, the definition of a software product is presented in Frame 1.6.

Frame 1.6: Software product definition

Software product is
A collection of components necessary to ensure proper operation, and efficient maintenance during its life cycle. The components include (1) computer programs ("code"), (2) documentation, (3) data necessary for its operation and maintenance (including standard test), and (4) procedures.

It should be noted that software quality assurance refers to the quality of all components of the software product, namely, the code, documentation, necessary operating and standard test data, and procedures. Moreover, the composition of software product components varies significantly according to the software development tools and methodology.

1.3 The principles of SQA

Source: after ISO 9000:2000 (ISO, 2000)

The following principles guide organizations in their process to ensure the software quality of their software products and services satisfies the needs and wants of stakeholders.

- **Customer focus.** Organizations depend on their customers, and thus need to understand their current and future needs, fulfill their requirements, and achieve their satisfaction.
- **Leadership.** An organization's leaders should create an internal environment in which employees are involved in achieving the quality targets.
- **Involvement of people-employees.** The involvement of employees at all levels enables benefiting from their capabilities to promote software quality issues.
- **Process approach.** Managing activities and resources as processes results in their improved efficiency.
- **System approach to management.** Process management achieves higher effectiveness and efficiency through identification, analysis, and understanding of interrelated processes.
- **Continual improvement.** Continual combined improvement of quality and processes' effectiveness and efficiency performance are a permanent objective of the organization.
- **Factual approach of decision-making.** Decisions should be based on data and information.
- **Mutually beneficial supplier relationships.** Understanding that an organization's supplier relationships based on mutual benefits contributes to improved performance of the organization with regard to quality, efficiency, and effectiveness.

1.4 Software errors, faults, and failures

To better understand the essence of software errors, faults, and failures, let us take a look at the performance of a deployed software system, as perceived by customers.

Example: The Simplex HR is a software system that has been on the market for 7 years. Its software package currently serves about 1200 customers.

One of the staff from the Simplex HR Support Centre reported a number of quotes from typical customer complaints:

1. "We have been using the Simplex HR software in our Human Resources Department for about four years, and have never experienced a software failure. We have recommended the Simplex HR to our colleagues."
2. Immediately following this positive testimony, the same employee complained that he could not prepare a simple monthly report.
3. "I started to use the Simplex HR two months ago; we have experienced so many failures that we are considering replacing the Simplex-HR software package."
4. "We have been using the software package for almost five years, and were very satisfied with its performance, until recently. During the last few months, we suddenly found ourselves having to contend with several severe failures."

Is such a variation in user experience relating to failures possible for the very same software package?

Can a software package that successfully served an organization for a long period of time "suddenly" change its nature (quality) and be full of bugs?

The answer to both these questions is YES, and the reason for this is rooted in the very characteristics of software errors.

The origin of software failures lies in a *software error* made by a software designer or programmer. An error may refer to a grammatical error in one or more of the code lines, or a logical error in carrying out one or more of the specification requirements.

A *software fault* is a software error that causes improper functioning of the software in a specific application, and in rare cases, of the software in general. However, not all software errors become software faults. In many other cases, erroneous code lines will not affect the functionality of the software (software faults are not caused). It should be noted that in some software fault cases, the fault is corrected or "neutralized" by subsequent code lines.

Naturally, our interest lies mainly in *software failures* that disrupt the use of the software. A software failure is a result of a software fault, hence our next question.

Do all software faults inevitably cause software failures? Not necessarily: A software fault becomes a software failure only when it is "activated" – that is when the software user tries to apply the specific, faulty application. In many cases, a software fault is in fact never activated. This is either due to the user's lack of interest in the specific application, or to the fact that the combination of conditions necessary to activate the software fault never occurs. The following two examples demonstrate the software fault – software failure relationships.

Example 1 The Simplex HR software package

Let us return to the Simplex HR software package mentioned above.

The software package includes the following fault:

1. Overtime compensation – This function was defined to allow two levels of daily overtime, where the user can specify the details and compensation per each level. For instance, the first 2 hours' overtime (level 1) should be paid at a rate that is 25% more than the regular hourly rates, while each following additional hour (level 2) should be paid at a rate that is 50% more than the regular hourly rates.

   The programmer's mistake caused the following fault: In cases when two levels of overtime were reported, the higher compensation was paid for overtime hours reported for both the levels.

Let us now examine the software failures experienced by two of Simplex HR users:

a. **A chain of pharmacies**

   Overtime pay – The policy of the chain was to implement overtime for no more **than 2 hours on top. The first level of overtime compensation was defined at 3** hours.

   Thanks to its policy, the chain did not experience software failures relating to the overtime features?

b. **A regional school**

   Overtime pay – The school has lately introduced the Simplex HR software package to support the management of its teacher staff. Cases of overtime happen quite frequently, and are due to the replacement of teachers on sick leave, personal leave of absence, and so on. The teachers' compensation was 30% above their hourly regular rate for the first 2 hours (level 1), and 75% above their hourly rate per each additional hour overtime (level 2). The failure related to overtime calculations was evident from the first salary calculations. Teachers who worked relatively long hours' overtime (over 2 hours per time) in the past months were both astonished and delighted to discover significantly higher overtime compensation than anticipated.

It should be noted that once software failures are identified, Simplex HR maintenance team is expected to correct them.

Example 2 The "Meteoro-X" meteorological equipment firmware

Meteoro-X is a computerized recording and transmission equipment unit designed for meteorological stations that perform temperature and precipitation measurements. The Meteoro-X is also equipped with three wind vanes for wind

velocity measurements. Meteorological measurements are defined to be transmitted every 5 minutes to a meteorological center.

"Meteoro-X" firmware (software embedded in the product) includes the following software fault:

Temperature threshold – The safety control specifications require shutting down the equipment if its temperature rises above 50 degrees centigrade.

The programmer error that resulted in a software fault – he registered the threshold as 150 degrees centigrade. This fault could only be noted, and consequently cause damage, when the equipment was subjected to temperatures measuring higher than 50 degrees.

Let us now examine the failure experienced by some of the Meteoro-X users:

a. Meteorological authorities of a southern European country

Temperature threshold – The Meteoro-X performed with no failures for about 3 years, due to the fact that temperatures higher than 50 degrees centigrade had not been recorded. It was only in the month of August of the fourth year when temperatures reached 57 degrees centigrade that an equipment disaster in one of the meteorological stations occurred.

b. North European Meteorological Board

Temperature threshold – The Meteoro-X had no failures due to the fact that temperatures higher than 50 degrees centigrade were not recorded.

A review of the specification document and the relevant code modules revealed the causes of the software faults, and enabled their correction.

These examples clearly demonstrate that at some time during the software service, some software faults will become software failures. Other software faults, and in some cases even a major portion of them, will remain hidden, invisible to software users, only to be activated when specific conditions are in place.

Figure 1.1 illustrates the relationships between software errors, faults, and failures; of the 17 software errors yielded in the development process, 8 become

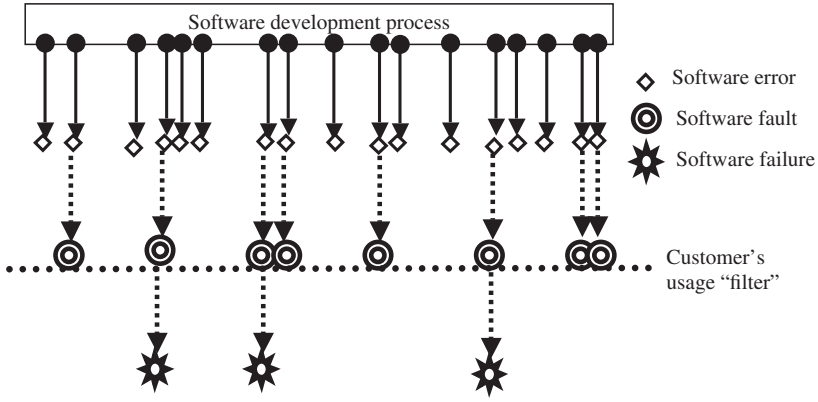

Figure 1.1 Software errors, software faults, and software failures

software faults, while only 3 of these faults become software failures. The customer's software usage characteristics determine which software applications are used, and thereby which faults become failures. In other words, the characteristics serve as a "failure filter."

1.5 The causes of software errors

As software errors are the cause of poor software quality, it is important to investigate their causes, in order to prevent them. It should be noted that these errors are all human errors, made by system analysts, programmers, software testers, documentation experts, managers, and sometimes clients and their representatives. Even in rare cases where software errors may be caused by the development environment: interpreters, wizards, automatic software generators, and so on, it is reasonable to claim that these too are human errors, as someone is responsible for the failure of the development. The causes of software errors can be classified according to the stages of the software development process in which they occur. A classification of error causes into nine classes is presented:

a. Faulty definition of requirements

A faulty definition of a requirement, usually prepared by the client, is one of the main causes of software errors. The most common errors of this type are:

- Erroneous definition of requirements
- Lack of essential requirements
- Incomplete requirements definition

  For instance, one of the requirements of a municipality's local tax software system refers to discounts granted to various segments of the population: senior citizens, parents of large families, and so on. Unfortunately, a discount granted to students was not included in the requirements document.
- Inclusion of unnecessary requirements, functions that are not expected to be applied.

b. Client–developer communication failures

Misunderstandings resulting from defective client–developer communication are additional causes for errors that prevail in the early stages of the development process:

- Misunderstanding of the client's instructions in the requirement document.
- Misunderstanding of the client's requirement changes presented to the developer in written form or verbally during the development period.
- Misunderstanding of the client's responses to design issues presented by the developer.
- Lack of attention to client messages relating to requirement changes, and client responses to questions raised by the developer.

c. Deliberate deviations from software requirements

In several circumstances, developers may deliberately deviate from the documented requirements – an action that often causes software errors. The most common situations of deliberate deviations are:

- Developer reuses software modules from previous project without sufficient analysis of the changes and adaptations needed to correctly fulfill all relevant customer requirements.
- Developer decides to omit part of the required functions in an attempt to better handle time or budget pressures.
- Developer-initiated improvements to the software introduced without managerial or client approval. Improvements of this type frequently disregard project requirements deemed minor by the developer. Such "minor" requirements when ignored create changes that may eventually cause software errors.

d. Logical design errors

Software errors can enter the system when professionals designing the system; system architects, software engineers, system analysts, and so on formulate the software requirements into design definitions. Typical logical errors include:

- Definitions that represent software requirements by means of erroneous algorithms.
- Process definitions that contain sequencing errors.

  Example: The software requirements for a firm's debt collection system define a debt collection process that includes the following requirement: Once a client, after receiving three successive notification letters, does not pay his debt; the client details are to be reported to the Sales Department Manager, who will decide whether to proceed to the next stage, which is referral of the client to the Legal Department. The system analyst defined the process incorrectly by stating that if no receipt of payment is noted after sending three successive letters, the client personal and debt details will be included on a list of clients delivered to the Legal Department. The logical error was caused by the analyst's erroneous omission of the Sales Department phase from the debt collection process.
- Erroneous definition of boundary conditions.

  Example: The client requirements stated that a special discount will be granted to customers who make more than three purchase transactions in the same month. The analyst erroneously defined the software process to state that the discount would be granted to those who make three or more transactions in the same year.
- Omission of required software system states.

  Example: Real-time computerized apparatus is required to respond in a specific way to a combination of temperatures and pressures. The

analyst did not define the required response when the temperature is over 120 degrees centigrade, and the pressure between 6 and 8 atmospheres.

- Omission of definitions concerning reactions to illegal operation of the software system.

  Example: A computerized theatre ticketing system operated by the customer has no human operator interface. The software system is required to limit sales to 10 tickets per customer. Accordingly, any request for the purchase of more than 10 tickets is "illegal." In the design, the analyst included a message stating that sales are limited to 10 tickets per customer, but did not define the system response to cases when customers (who might not have properly understood the message) key in a number higher than 10. When performing this illegal request, a system "crash" may be expected, as no computerized response was defined for this illegal operation.

e. Coding errors

A wide range of reasons cause programmers to make coding errors. These include misunderstanding the design documentation, linguistic errors in programming languages, errors in the application of CASE and other development tools, errors in data selection, and so on.

f. Noncompliance with documentation and coding instructions

Almost every development unit has its own documentation and coding standards that define the content, order and format of the documents, and code developed by team members. For this purpose, the unit develops and publicizes templates and coding instructions. Members of the development team or unit are required to comply with these directions.

As it may be assumed that errors of noncompliance with instructions do not usually become software faults, one may ask why cases of noncompliance with these instructions should be considered as software errors. Even if the quality of the "noncomplying" software is acceptable, difficulties will inevitably be presented when trying to understand it. In other words, future handling of this software (by development and/or maintenance teams) is expected to substantially increase the rate of errors in the following situations:

- Team members, who need to coordinate their own code with code modules developed by "noncomplying" team members, can be expected to encounter more difficulties than usual when trying to understand the software.
- Individuals replacing the "noncomplying" team member (who retired or was promoted) will find it difficult to fully understand the "noncomplying" code.

- The design review team will find it more difficult to study a design document prepared by a "noncomplying" team, and as a result will probably misunderstand part of the design details.
- The test team will find it more difficult to test the "noncomplying" module; consequently, their effectiveness is expected to be decreased, leaving more errors undetected. Moreover, team members required to correct the detected errors can be expected to encounter greater difficulties when doing so. They may leave some errors only partially corrected, and even introduce new errors as a result of their incomplete grasp of the other team member's work.
- Maintenance teams required to contend with "bugs" detected by users, and to change or add to the existing software will face extra difficulties when trying to understand the "noncomplying" software and its documentation. This is expected to result in an excessive number of errors, along with increased maintenance expenditures.

g. Shortcomings of the testing process

Shortcomings of the testing process affect the error rate by leaving a greater number of errors undetected or uncorrected. These shortcomings result from:

- Incomplete test plans failing to test all or some parts of the software, application functions, and operational states of the system.
- Failure to document and report detected errors and faults.
- Failure to promptly correct detected software faults, as a result of inappropriate indications of the reasons for the fault.
- Incomplete testing of software error corrections
- Incomplete corrections of detected errors due to negligence or time pressures.

h. User interface and procedure errors

User interfaces direct users in areas such as the performance of input and output activities, and data collection and processing. Procedures direct users with respect to the sequence of activities required at each step of the process. Procedures are of special importance in complex software systems, where processing is conducted in several steps, each of which may feed a variety of types of data and enable examination of intermediate results. User interface and procedure errors may cause processing failures even in cases of error-free design and coding. The following example presents a procedure error.

Example

"Eiffel," a construction material store, has decided to grant a 5% discount to major customers, who are billed monthly. The discount is offered to customers whose total net purchases in the store in the preceding 12 months exceeded $1 million. The discount is effective for the last

Table 1.1 "Eiffel" billing procedures – correct and incorrect discount procedures

Correct procedure	Incorrect procedure
At the beginning of each month, Eiffel's information processing department:	At the end of each month, Eiffel's information processing department:
1. Calculates the cumulative purchases for the last 12 months (A) and the cumulative returns for the last 12 months (B) for each of its major customers. **2.** Calculates the net cumulative purchases ($A-B$) for each major customer for the last 12 months in the store. **3.** For major customers, whose ($A-B$) > \$1 million and $B/(A-B)$ <10%, calculate 5% discount on their last month's account.	**1.** Calculates the cumulative purchases for the last 12 months (A) and the cumulative returns for the last 12 months (B) for each of its major customers. **2.** For major customers, whose A > \$1 million, and (B/A) <10%, calculate 5% discount on their last month's account.

month's account. Furthermore, the management decided to withdraw the discount from customers who returned goods valued in excess of 10% of their net purchases during the last 12 months.

Table 1.1 presents a comparison of correct and incorrect procedures regarding application of the discount.

It is clear that under the incorrect procedure, customers, whose net purchases ($A-B$) are equal or below \$1 million and/or their percentage of returned goods ($B/A-B$) is equal or exceeds 10%, may be mistakenly found to be eligible for the 5% discount

i. Documentation errors

The documentation errors of concern to the development and maintenance teams are those found in the design, software manuals, documents, and in the documentation integrated into the body of the software. These errors can cause additional errors in further stages of development and during the maintenance period.

Another type of documentation errors that affect mainly users are errors in the user manuals and in the "help" displays incorporated in the software. Typical errors of this type are:

- Omission of software functions.
- Errors in the explanations and instructions given to users, resulting in "dead ends" or incorrect applications.
- Listings of nonexisting software functions, usually functions planned in the early stages of development but later dropped, but also functions that

were active in previous versions of the software but cancelled in the current version.

Frame 1.7 summarizes the causes of software errors.

Frame 1.7: The nine causes of software errors

The nine causes of software errors are:
a. Faulty requirements definition **b.** Client–developer communication failures **c.** Deliberate deviations from software requirements **d.** Logical design errors **e.** Coding errors **f.** Noncompliance with documentation and coding instructions **g.** Shortcomings of the testing process **h.** User interface and procedure errors **i.** Documentation errors Additional approaches to classification of software defects and their causes are presented by Ko and Myers (2005) and Thung et al. (2012).

1.6 Software quality assurance versus software quality control

Two terms are constantly repeated within the context of software quality: "software quality control" and "software quality assurance." Are they synonymous? How are they related?

Definitions of software quality assurance are already presented in Frames 1.2 and 1.3. In order to compare the two terms, definitions for SQC are presented in Frame 1.8.

Frame 1.8: Software quality control – the IEEE definitions

Source: IEEE Std. 610.12-1990 (IEEE, 1990)

Software quality control is
1. A set of activities designed to evaluate the quality of a developed or manufactured product. Contrast with software quality assurance. **2.** The process of verifying one's own work or that of coworker.

SQA and SQC represent two distinct concepts.

Software quality control relates to the activities needed to evaluate the quality of a final software product, with the main objective of withholding any product that does not qualify. In contrast, the main objective of **software quality assurance** is to minimize the cost of ensuring the quality of a software product with a variety of infrastructure activities and additional activities performed throughout the software development and maintenance processes/stages. These activities are aimed at preventing the causes of errors, and at detecting and correcting errors that may have occurred at the earliest possible stage, thus bringing the quality of the software product to an acceptable level. As a result, quality assurance activities reduce substantially the probability that software products will not qualify and, at the same time, in most cases, reduce the costs of ensuring quality.

In summary,

1. SQC and SQA activities serve different objectives.
2. SQC activities are only a part of the total range of SQA activities.

1.7 Software quality engineering and software engineering

The definition of **software engineering**, according to the IEEE, is presented in Frame 1.9.

Frame 1.9: Software engineering – the IEEE definition

Source: IEEE Std. 610.12-1990

Software engineering is
The application of a systematic, disciplined, quantifiable approach to the development, operation, and maintenance of software, that is, the application of engineering to software.

The characteristics of **software engineering**, especially those of the systematic, disciplined, and quantitative approach at its core, make it a good infrastructure for achieving effective and efficient software development and maintenance objectives. The methodologies and tools applied by software engineering determine the process of transforming a software requirement document into a software product, and also include the performance of quality assurance activities. **Software quality engineering** employs the development of quality assurance methodologies, procedures, and tools together with methods for follow-up of quality assurance activities performed by software development and maintenance teams.

Software quality engineering and software engineering have a great number of topics in common. Albeit the two groups view these topics from different standpoints – respective to their profession, their shared knowledge and cooperation are the basis for successful software development.

An indication of the extent of shared topics may be perceived when comparing the software engineering body of knowledge (SWEBOK) (Bourque and Fairley, 2014) and the certified software quality engineer body of knowledge (CSQEBOK) (ASQ, 2016). A detailed discussion of an earlier version of the CSQEBOK was compiled by Westfall (2009).

Summary

1. **Definitions of software, software quality, and software quality assurance**
   Software, from the SQA perspective, is the combination of computer programs ("code"), procedures, documentation, and data necessary for operating the software system. The combination of all four components is needed to ensure the quality of the development process, as well to ensure quality during extended maintenance periods.
   Software quality, according to Pressman's definition, is the degree of conformance to specific functional requirements, specified software quality standards, and Good Software Engineering Practices (GSEP).
   Software quality assurance. This book adopts an expanded definition of the widely accepted IEEE definition of software quality assurance. Accordingly, software quality assurance is the systematically planned set of actions necessary to provide adequate confidence that a software development, or maintenance process, conforms to established functional technical requirements, and also to the managerial requirements of keeping to schedule and operating within budget.
2. **The distinction between software errors, software faults, and software failures**
   Software errors are sections of the code that are partially or totally incorrect as a result of a grammatical, logical, or other type of mistake made by a system analyst, programmer, or other member of the software development team.
   Software faults are software errors that cause the incorrect functioning of the software during one of its specific applications.
   Software faults become **software failures** only when they are "activated," that is, when a user tries to apply the specific software section that is faulty. Thus, the root of any software failure is a software error.

3. **The various causes of software errors**

   There are nine causes of software errors: (1) faulty requirements definition, (2) client–developer communication failures, (3) deliberate deviations from software requirements, (4) logical design errors, (5) coding errors, (6) noncompliance with documentation or coding instructions, (7) shortcomings of the testing process, (8) procedure errors, and (9) documentation errors. It should be emphasized that all errors are human errors, and are made by system analysts, programmers, software testers, documentation experts, and even clients and their representatives.

4. **The objectives of software quality assurance activities**

   The objectives of SQA activities for software development and maintenance are:

   1. Ensuring, with acceptable levels of confidence, conformance to functional technical requirements.
   2. Ensuring, with acceptable levels of confidence, conformance to managerial requirements of scheduling and budgets.
   3. Initiating and managing activities for the improvement and greater efficiency of software development and SQA activities.

5. **The differences between software quality assurance and software quality control**

   Software quality control is a set of activities carried out with the main objective of withholding software products from delivery to the client if they do not qualify. In contrast, the objective of software quality assurance is to minimize the costs of software quality by introducing a variety of infrastructure activities and other activities throughout the development and maintenance processes. These activities are performed in all stages of development to eliminate causes of errors, and detect and correct errors in the early stages of software development. As a result, quality assurance substantially reduces the rate of nonqualifying products.

6. **The relationship between software quality assurance and software engineering**

   Software engineering is the application of a systematic, disciplined, quantifiable approach to the development, operation, and maintenance of software.

   Software quality assurance practices are intertwined with the software engineering process in several ways: (1) SQA considerations affect the choice of software development tools and procedures. (2) SQA activities, such as design reviews and software tests, are incorporated in the software development activities. (3) SQA participation in the development of the software

development infrastructure of procedures, staff training, configuration management, and so on.

Selected bibliography

ASQ (2016) *The Certified Software Quality Engineering Body of Knowledge (CSQE BoK)*, American Society for Quality.

Bourque P. and Fairley R. (Eds.) (2014) *Guide to the Software Engineering Body of Knowledge SWEBOK*, Ver. 3.0, IEEE and IEEE Computer Society Press, Piscataway, NJ.

IEEE (1990) *IEEE Std. 610.12-1990-IEEE Standard Glossary of Software Engineering Terminology*, Corrected Edition, in IEEE, *IEEE Standards Collection*, The Institute of Electrical and Electronics Engineering, New York.

IEEE (2014) *IEEE Std. 730-2014 Software Quality Assurance*, The IEEE Computer Society, IEEE, New York.

ISO (2000) *ISO Std. 9000:2000 – Quality Management Systems – Fundamental and Vocabulary*, International Organization for Standardization (ISO), Geneva, Switzerland.

ISO/IEC/IEEE (2014) *ISO/IEC 90003:2014 – Software Engineering – Guidelines for the Application of ISO 9001:2008 to Computer Software*, International Organization for Standardization (ISO), Geneva, Switzerland.

Ko, A. J. and Myers, B. A. (2005) *A framework and methodology for studying the causes of software errors in programming systems*, *Journal of Visual Languages and Computing*, Vol. **16**, pp. 41–84.

Thung F., Lo, D., Jiang, L. (2012) *Automatic defect categorization, in* Proceedings of the 19th Working Conference on Reverse Engineering, pp. 205–214.

Westfall L. (2009) *The Certified Software Quality Engineer Handbook*, ASQ Quality Press, Milwaukee, WI.

Review questions

1.1 A software product comprises four main components.

- **a.** List the four components of a software system.
- **b.** How does the quality of each component contribute to the quality of the developed software?
- **c.** How does the quality of each component contribute to the quality of the software maintenance?

1.2 Refer to the following terms: software error, software fault, and software failure.

- **a.** Define the terms.
- **b.** Explain the differences between these undesirable software issues.
- **c.** Suggest a case where in a software package serving 300 clients, a new software failure ("bug") appears for the first time 6 years after the software package was first sold to the public.

1.3 Consider the principles of SQA

a. Explain in your own words the importance of the 6th principle.

b. How can the implementation of the 8th principle contribute to the quality of software product?

1.4 **a.** List and briefly describe the various causes of software errors.

b. Classify the causes of errors according to the group/s responsible for the error – the client staff, the system analysts, the programmers, the testing staff – or is the responsibility a shared one, belonging to more than one group?

1.5 What are the differences between the IEEE definition of SQA and the expanded definition discussed in this book?

1.6 According to the IEEE definition of SQC, SQC is in contrast with SQA.

a. In what respect does SQC vary from SQA?

b. In what way can SQC be considered part of SQA?

Topics for discussion

1.1 A programmer claims that as only a small proportion of software errors turn into software failures, it is unnecessary to make substantial investments in the prevention and elimination of software errors.

a. Do you agree with this view?

b. Discuss the outcome of accepting this view.

1.2 George Wise is an exceptional programmer. Testing his software modules reveals very few errors, much less than the team's average. He is very rarely late in completing a task. George always finds original ways to solve programming challenges, and uses an original, individual version of the coding style. He dislikes preparing the required documentation, and rarely does so according to the team's templates.

A day after completing a challenging task, on time, he was called to the office of the department's chief software engineer. Instead of being praised for his accomplishments (as he expected), he was warned by the company's chief software engineer that he would be fired, unless he began to fully comply with the team's coding and documentation instructions.

a. Do you agree with the position taken by the department's chief software engineer?

b. If you agree, could you suggest why his/her position was so decisive?

c. Explain how George's behavior could cause software errors.

1.3 The claim, according to the expanded definition of SQA, that a development team should invest its efforts equally for complying with project requirements as they invest in keeping project schedule and budget supports client satisfaction.

a. Do you agree with this claim?

b. If yes, provide arguments to substantiate your position.

1.4 Five reasons for shortcomings of the testing process are mentioned in Section 1.5.

a. Explain these five reasons in your own words.

b. Could you suggest circumstances of a testing process which could cause these shortcomings?