Chapter 1

Software Quality Fundamentals

After completing this chapter, you will be able to:

- use the correct terminology to discuss software quality issues;
- identify the major categories of software errors;
- understand the different viewpoints regarding software quality;
- provide a definition of software quality assurance;
- understand software business models as well as their respective risks.

1.1 INTRODUCTION

Software is developed, maintained, and used by people in a wide variety of situations. Students create software in their classes, enthusiasts become members of open-source development teams, and professionals develop software for diverse business fields from finance to aerospace. All these individual groups will have to address quality problems that arise in the software they are working with. This chapter will provide definitions for terminology and discuss the source of software errors and the choice of different software engineering practices depending on an organization's sector of business.

Every profession has a body of knowledge made up of generally accepted principles. In order to obtain more specific knowledge about a profession, one must either: (a) have completed a recognized curriculum or (b) have experience in the domain. For most software engineers, software quality knowledge and expertise is acquired in a hands-on fashion in various organizations. The Guide to the Software Engineering Body of Knowledge (SWEBOK) [SWE 14] constitutes the first international consensus developed on the fundamental knowledge required by all software engineers.

Software Quality Assurance, First Edition. Claude Y. Laporte and Alain April.

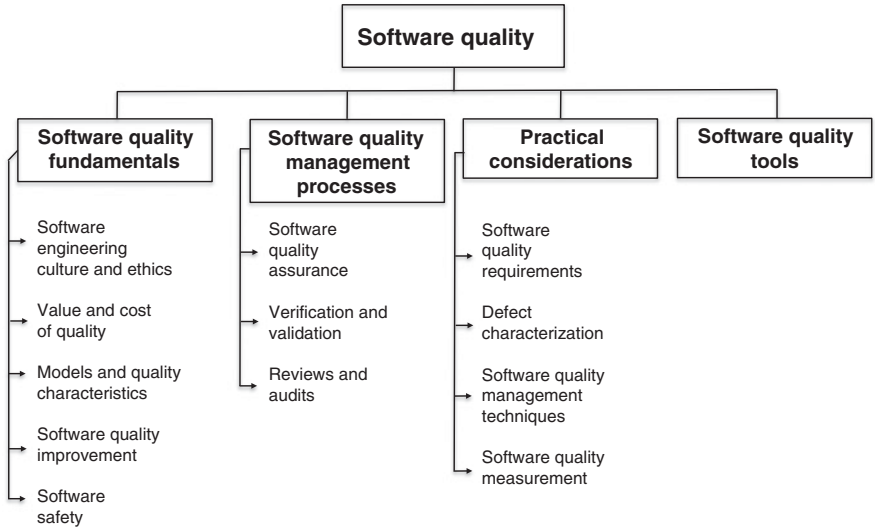


Figure 1.1 Software Quality in the SWEBOK® Guide [SWE 14].

Chapter 10 of SWEBOK is dedicated to software quality (see Figure 1.1) and its first topic is labeled "fundamentals" and introduces the concepts and terminology that form the underlying basis for understanding the role and scope of software quality activities. The second topic refers to the management processes and highlights the importance of software quality across the life cycle of a software project. The third topic presents practical considerations where various factors that influence planning, management, and selection of software quality activities and techniques are discussed. Last, software quality related tools are presented.

1.2 DEFINING SOFTWARE QUALITY

Before explaining the components of software quality assurance (SQA), it is important to consider the basic concepts of software quality. Once you have completed this section, you will be able to:

- define the terms "software," "software quality," and "software quality assurance";
- differentiate between a software "error," a software "defect," and a software "failure."

Intuitively, we see software simply as a set of instructions that make up a program. These instructions are also called the software's source code. A set of programs

forms an application or a software component of a system with hardware components. An information system is the interaction between the software application and the information technology (IT) infrastructure of the organization. It is the information system or the system (e.g., digital camera) that clients use.

Is ensuring the quality of the source code sufficient for the client to be able to obtain a quality system? Of course not; a system is far more complex than a single program. Therefore, we must identify all components and their interactions to ensure that the information system is one of quality. An initial response to the challenge regarding software quality can be found in the following definition of the term "software."

Software

1) All or part of the programs, procedures, rules, and associated documentation of an information processing system.

2) Computer programs, procedures, and possibly associated documentation and data pertaining to the operation of a computer system.

ISO 24765 [ISO 17a]

When we consider this definition, it is clear that the programs are only one part of a set of other products (also called intermediary products or software deliverables) and activities that are part of the software life cycle.

Let us look at each part of this definition of the term "software" in more detail:

- Programs: the instructions that have been translated into source code, which have been specified, designed, reviewed, unit tested, and accepted by the clients;
- Procedures: the user procedures and other processes that have been described (before and after automation), studied, and optimized;
- Rules: the rules, such as business rules or chemical process rules, that had to be understood, described, validated, implemented, and tested;
- Associated documentation: all types of documentation that is useful to customers, software users, developers, auditors, and maintainers. Documentation enables different members of a team to better communicate, review, test, and maintain software. Documentation is defined and produced throughout the key stages of the software life cycle;
- Data: information that is inventoried, modeled, standardized, and created in order to operate the computer system.

Software found in embedded systems is sometimes called microcode or firmware. Firmware is present in commercial mass-market products and controls machines and devices used in our daily lives.

Firmware
Combination of a hardware device and computer instructions or computer data that reside as read-only software on the hardware device.
ISO 24765 [ISO 17a]

1.3 SOFTWARE ERRORS, DEFECTS, AND FAILURES

If you listen closely during various meetings with your colleagues, you will notice that there are many terms that are used to describe problems with a software-driven system. For example:

- The system crashed during production.
- The designer made an error.
- After a review, we found a defect in the test plan.
- I found a bug in a program today.
- The system broke down.
- The client complained about a problem with a calculation in the payment report.
- A failure was reported in the monitoring subsystem.

Do all of these terms refer to the same concept or to different concepts? It is important to use clear and precise terminology if we want to provide a specific meaning to each of these terms. Figure 1.2 describes how to use these terms correctly.

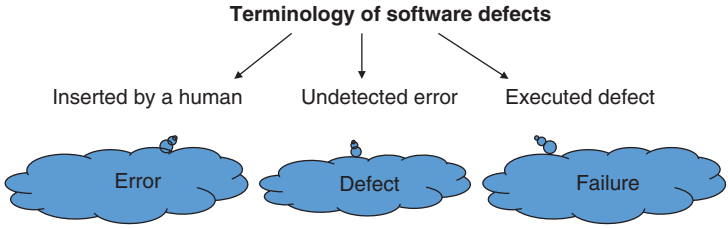


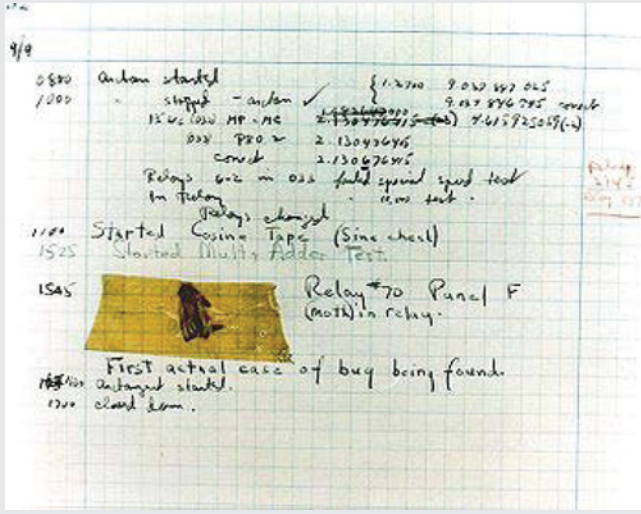
Figure 1.2 Terminology recommended for describing software problems.

A

Bug

Since the time of Thomas Edison, engineers have used the word "bug" to refer to failures in the systems that they have developed. This word can describe a multitude of possible problems. The first documented case of a "computer bug" involved a moth trapped in a relay of the Mark II computer at Harvard University in 1947. Grace Hopper, the computer operator, pasted the insect into the laboratory log, specifying it as the "First actual case of a bug being found" (see the page of this log in the photograph below).

In the early 1950s, the terms "bug," "debug," and "debugging," as applied to computers and computer programs, started to appear in the popular press [KID 98].



Photograph from the Smithsonian National Museum of American History.

A failure (synonymous with a crash or breakdown) is the execution (or manifestation) of a fault in the operating environment. A failure is defined as the termination of the ability of a component to fully or partially perform a function that it was designed to carry out. The origin of a failure lies with a defect hidden, that is, not detected by tests or reviews, in the system currently in operation. As long as the system in production does not execute a faulty instruction or process faulty data, it will run normally. Therefore, it is possible that a system contains defects that have not yet been executed. Defects (synonym of faults) are human errors that were not detected during software development, quality assurance (QA), or testing. An error can be found in the documentation, the software source code instructions, the logical execution of the code, or anywhere else in the life cycle of the system.

Error, Defect, and Failure

Error

A human action that produces an incorrect result (ISO 24765) [ISO 17a].

Defect

1) A problem (synonym of fault) which, if not corrected, could cause an application to either fail or to produce incorrect results. (ISO 24765) [ISO 17a].

2) An imperfection or deficiency in a software or system component that can result in the component not performing its function, e.g. an incorrect data definition or source code instruction. A defect, if executed, can cause the failure of a software or system component (ISTQB 2011 [IST 11]).

Failure

The termination of the ability of a product to perform a required function or its inability to perform within previously specified limits (ISO 25010 [ISO 11i]).

Figure 1.3 shows the relationship between errors, defects, and failures in the software life cycle. Errors may appear during the initial feasibility and planning stages of new software. These errors become defects when documents have been approved and the errors have gone unnoticed. Defects can be found in both intermediary products (such as requirements specifications and design) and the source code itself. Failures occur when an intermediary product or faulty software is used.

Case of Errors, Defects, and Failures

Case 1: A local pharmacy added a software requirement to its cash register to prevent sales of more than $75 to customers owing more than $200 on their pharmacy credit card. The programmer did not fully understand the specification and created a sales limit of $500 within the program. This defect never caused a failure since no client could purchase more than $500 worth of items given that the pharmacy credit card had a limit of $400.

Case 2: In 2009, a loyalty program was introduced to the clients of American Signature, a large furniture supplier. The specifications described the following business rules: a customer who makes a monthly purchase that is higher than the average amount of monthly purchases for all customers will be considered a Preferred Customer. The Preferred Customer will be identified when making a purchase, and will be immediately given a gift or major discount once a month. The defect introduced into the system (due to a poor understanding of the algorithm to set up for

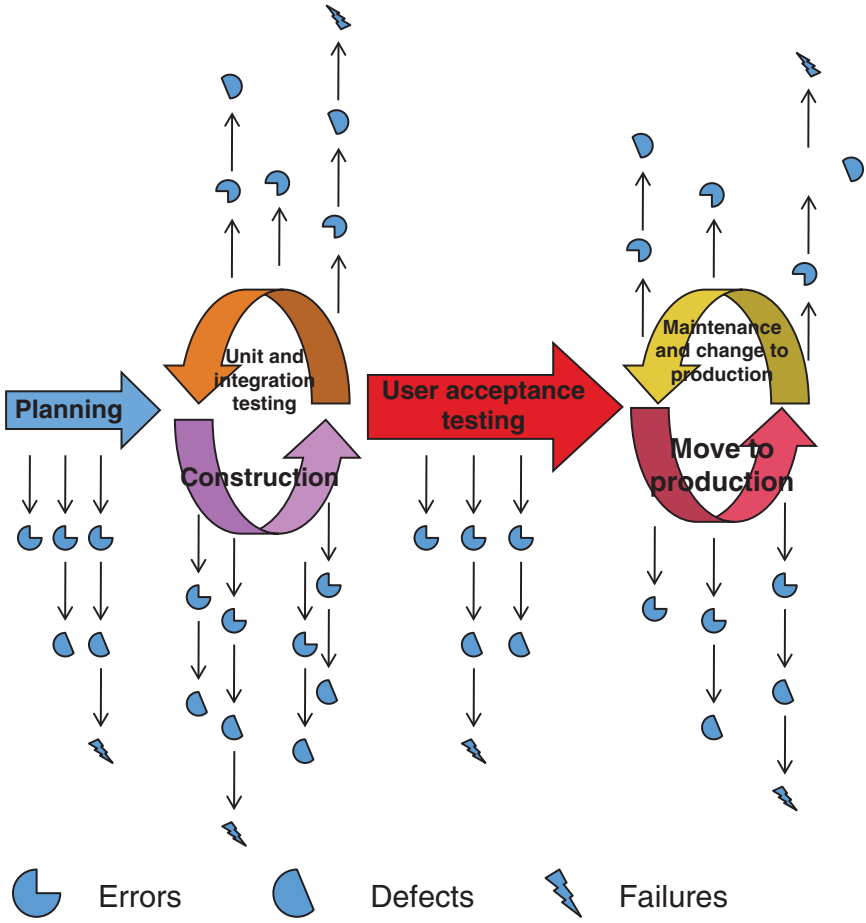


Figure 1.3 Errors, defects, and failures in the software life cycle.
Source: Galin (2017). [GAL 17]. Adapted with permission of Wiley-IEEE Computer Society Press.

this requirement) involved only taking into account the average amount of current purchases and not the customer's monthly history. At the time of the software failure, the cash register was identifying far too many Preferred Clients, resulting in a loss for the company.

Case 3: Peter tested Patrick's program when Patrick was away. He found a defect in the calculation for a retirement savings plan designed to apply the new tax-exemption law for this type of investment. He traced the error back to the project specification and informed the analyst. In this case, the test activity correctly identified the defect and the source of the error.

The three cases above correctly use the terms to describe software quality problems. They also identify issues that are investigated by researchers in the field of software quality in order to discover means to help eliminate these problems:

- Errors can occur in any of the software development phases throughout the life cycle.
- Defects must be identified and fixed before they become failures.
- The cause of failures, defects, and errors must be identified.

Life Cycle

Evolution of a system, product, service, project, or other human-made entity from conception through retirement.

Development Life Cycle

Software life cycle process that contains the activities of requirements analysis, design, coding, integration, testing, installation, and support for acceptance of software products.

ISO 12207 [ISO 17]

During software development, defects are constantly being involuntarily introduced and must be located and corrected as soon as possible. Therefore, it is useful to collect and analyze data on the defects found as well as the estimated number of defects left in the software. By doing so, we can improve the software engineering processes and in turn, minimize the number of defects introduced in new versions of software products in the future.

Methods for classifying defects have been created for this purpose, one of which is explained in the chapter on verification and validation.

Undetected Hole in the Ozone Layer

The hole in the ozone layer over Antarctica went unnoticed for a long period of time because the TOMS data analysis software used by NASA as part of its project to map the ozone layer had been designed to ignore values that deviate significantly from the anticipated measurements.

The project was launched in 1978, but it was only in 1985 that the hole was discovered, and not by NASA. Following data analysis, NASA confirmed this design error.

http://earthobservatory.nasa.gov/Features/RemoteSensingAtmosphere/remote_sensing5.php

Depending on the business model of your organization, you will have to allow for varying degrees of effort in identifying and correcting defects. Unfortunately, there exists today a certain culture of tolerance for software defects. However, there is no question that we all want Airbus, Boeing, Bombardier, and Embraer to have identified and corrected all the defects in the software for their airplanes before we board them!

Many researchers have studied the source of software errors and have published studies classifying software errors by type in order to evaluate the frequency of each type of error. Beizer (1990) [BEI 90] provides a study that has combined the result of several other studies to provide us with an indication of the origin of errors. The following is a summarized list of this study's results [BEI 90].

- 25% structural;
- 22% data;
- 16% functionalities implemented;
- 10% construction/coding;
- 9% integration;
- 8% requirements/functional specifications;
- 3% definition/running tests;
- 2% architecture/design;
- 5% unspecified.

Researchers also try to determine how many errors can be expected in a typical software. McConnell (2004) [MCC 04] suggested that this number varied based on the quality and maturity of the software engineering processes as well as the training and competency of the developers. The more mature the processes are, the fewer errors are introduced into the development life cycle of the software. Humphrey (2008) [HUM 08] also collected data from many developers. He found that a developer involuntarily creates about 100 defects for each 1000 lines of source code written. In addition, he noted large variations for a group of 800 experienced developers, that is, from less than 50 defects to more than 250 defects injected per 1000 lines of code. At Rolls-Royce, the manufacturer of airplane engines, the variation published is from 0.5 to 18 defects per 1000 lines of source code [NOL 15]. The use of proven processes, competent and well-trained developers, and the reuse of already proven software components can considerably reduce the number of errors of a software.

McConnell also referenced other studies that have come to the following conclusions:

- The scope of most defects is very limited and easy to correct.
- Many defects occur outside of the coding activity (e.g., requirement, architecture activities).

- Poor understanding of the design is a recurrent problem in programming error studies.
- It is a good idea to measure the number and origin of defects in your organization to set targets for improvement.

Therefore, errors are the main cause of poor software quality. It is important to look for the cause of error and identify ways in which to prevent these errors in the future. As we have shown in Figure 1.3, errors can be introduced at each step of the software life cycle: errors in the requirements, code, documentation, data, tests, etc. The causes are almost always human mistakes made by clients, analysts, designers, software engineers, testers, or users. SQA will need to develop a classification of the causes of software error by category which can be used by everyone involved in the software engineering process.

For example, here are eight popular error-cause categories:

1) problems with defining requirements;

2) maintaining effective communication between client and developer;

3) deviations from specifications;

4) architecture and design errors;

5) coding errors (including test code);

6) non-compliance with current processes/procedures;

7) inadequate reviews and tests;

8) documentation errors.

Each of the eight categories of error causes listed above is described in more detail in the following sections.

1.3.1 Problems with Defining Requirements

Defining software requirements is now considered a specialty, which means a business analyst or a software engineer specialized in requirements. Requirements definition is the topic of interest groups as well as the subject of professional certification programs (see http://www.iiba.org).

There are a certain number of problems related to the clear, correct, and concise writing of requirements so that they can be converted into specifications that can be directly used by colleagues, such as architects, designers, programmers, and testers.

It must also be understood that there are a certain number of activities that must be mastered when eliciting requirements:

- identifying the stakeholders (i.e., key players) who must participate in the requirements elicitation;
- managing meetings;
- interview techniques that can identify differences between wishes, expectations, and actual needs;

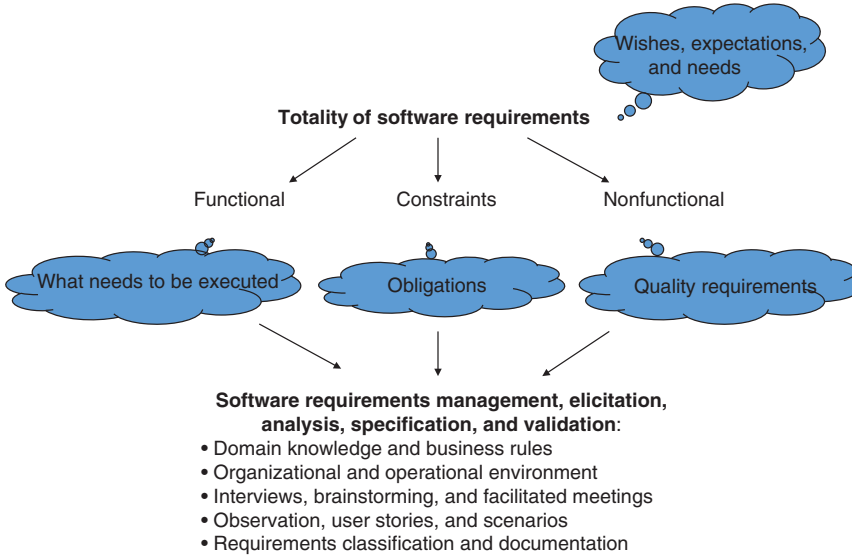


Figure 1.4 Context of software requirements elicitation.

- clear and concise documentation of functional requirements, performance requirements, obligations, and properties of future systems;
- applying systematic techniques for requirement elicitation;
- managing priorities and changes (e.g., changes to requirements).

It is clear that errors can arise when eliciting requirements. It can be difficult to cater to the wishes, expectations, and needs of many different user groups at the same time (see Figure 1.4). Therefore, it is important to pay particular attention to erroneous requirement definitions, the lack of definitions for critical obligations and software characteristics, the addition of unnecessary requirements (e.g., those not requested by the customer), the lack of attention to business priorities, and fuzzy requirements descriptions.

Ordering Soup

Let us say you order soup at a restaurant. Your expressed requirement is "I would like the soup of the day." But in fact, unexpressed wishes, expectations, and needs include: not too hot, not too cold; soup that is not too salty; utensils, salt, and pepper available on the table; clean washrooms; a well situated table; a quiet environment.

And that was a simple requirement, imagine a complex software!

A requirement is said to be of good quality when it meets the following characteristics:

- correct;
- complete;
- clear for each stakeholder group (e.g., the client, the system architect, testers, and those who will maintain the system);
- unambiguous, that is, same interpretation of the requirement from all stakeholders;
- concise (simple, precise);
- consistent;
- feasible (realistic, possible);
- necessary (responds to a client's need);
- independent of the design;
- independent of the implementation technique;
- verifiable and testable;
- can be traced back to a business need;
- unique.

We will present techniques to help detect defects in requirements documentation in a later chapter concerning reviews.

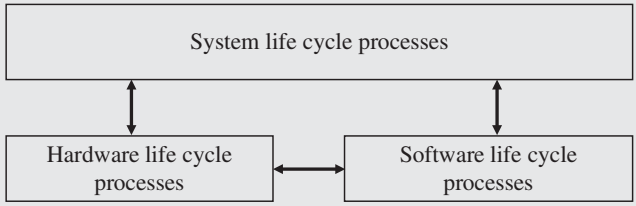
We must also ensure that we are not looking for the Holy Grail of the perfect specification, since we do not always have the time or means, or the budget, to achieve this level of perfection.

The article by Ambler [AMB 04] entitled "Examining the Big Requirements Up Front Approach" suggests that it is sometimes ineffective to write detailed requirements early in the life cycle of a software project. He claims that this traditional approach increases the risk of a project's failure. He stipulates that a large percentage of these specifications are not integrated in the final version of the software and that the corresponding documentation is rarely updated during the project. He thus asserts that this way of working is outdated. In his article, he recommends using more recent agile techniques, such as Test-Driven Development, in order to produce a minimum amount of paper documentation.

We have observed that software analysts and designers also often use prototyping, which helps to partially eliminate the traditional requirements document and replace it with a set of user interfaces and test cases that describe the requirements, architecture, and software design to be developed. Prototypes prove useful for pinpointing what the client is envisioning and getting valuable feedback early in the project. In the next section, the development practices adopted by different business sectors will be discussed.

In a system with hardware and software components, requirements are developed at the system level and then allocated to hardware, software, and sometimes to an operator. The following figure illustrates the interactions between system, hardware, and software life cycle processes.

Systems engineering must work in close collaboration with hardware and software engineering during the allocation of system requirements (for example, functionalities and quality requirements such as safety and performance) to hardware and software.



1.3.2 Maintaining Effective Communications Between Client and Developer

Errors can also occur in intermediary products due to involuntary misunderstandings between software personnel and clients and users from the outset of the software project. Software developers and software engineers must use simple, non-technical language and try to take into account the user's reality. They must be aware of all signs of lack of communication, on both sides. Examples of these situations are:

- poor understanding of the client's instructions;
- the client wants immediate results;
- the client or the user does not take the time to read the documentation sent to him;
- poor understanding of the changes requested from the developers during design;
- the analyst stops accepting changes during the requirements definition and design phase, given that for certain projects 25% of specifications will have changed before the end of the project.

To minimize errors:

- take notes at each meeting and distribute the minutes to the entire project team;
- review the documents produced;

- be consistent with your use of terms and develop a glossary of terms to be shared with all stakeholders;
- inform clients of the cost of changing specifications;
- choose a development approach that allows you to accept changes along the way;
- number each requirement and implement a change management process (as it will be presented in another chapter).

This book includes a glossary that could be used to develop the glossary for a specific project.

HVCCR is a company dedicated to the online sale of specialized ventilation, refrigeration, and air-conditioning tools. A client contacted the company that maintained its web catalog to update a few images and add some recent products. The task was estimated to take 10–20 minutes of work.

The person in charge of maintaining the web catalog contacted the client and informed him that to complete the update of the changes requested, the server would have to be restarted, which could cancel any current sessions. This work should preferably be done at night. The client, who did not fully grasp the impact of his request, insisted on having this update performed immediately. At the time of the update, several buyers were online processing payments. Shutting down the system interrupted bank transactions, causing customer dissatisfaction as well as data corruption. HVCCR took several days to address buyers' complaints and fix the problems.

1.3.3 Deviations from Specifications

This situation occurs when the developer incorrectly interprets a requirement and develops the software based on his own understanding. This situation creates errors that unfortunately may only be caught later in the development cycle or during the use of the software.

Other types of deviations are:

- reusing existing code without making adequate adjustments to meet new requirements;

– deciding to drop part of the requirements due to budget or time pressures;
– initiatives and improvements introduced by developers without verifying with clients.

1.3.4 Architecture and Design Errors

Errors can be inserted in the software when designers (system and data architects) translate user requirements into technical specifications. The typical design errors are:

– an incomplete overview of the software to be developed;
– unclear role for each software architecture component (responsibility, communication);
– unspecified primary data and data processing classes;
– a design that does not use the correct algorithms to meet requirements;
– incorrect business or technical process sequence;
– poor design of business or process rule criteria;
– a design that does not trace back to requirements;
– omission of transaction statuses that correctly represent the client's process;
– failure to process errors and illegal operations, which enables the software to process cases that would not exist in the client's sector of business—up to 80% of program code is estimated to process exceptions or errors.

1.3.5 Coding Errors

Many errors can occur in the construction of software. McConnell (2004) [MCC 04] devotes a substantial part of his book "*Code Complete"* to describing effective techniques for creating quality source code. He describes common programming errors and inefficiencies. According to McConnell, the typical programming errors are:

– inappropriate choice of programming language and conventions;
– not addressing how to manage complexity from the onset;
– poor understanding/interpretation of design documents;
– incoherent abstractions;
– loop and condition errors;
– data processing errors;
– processing sequence errors;
– lack of or poor validation of data upon input;

- poor design of business rule criteria;
- omission of transaction statuses that are required to truly represent the client process;
- failure to process errors and illegal operations, which enables the software to process cases that would not exist in the client's sector of business;
- poor assignment or processing of the data type;
- error in loop or interfering with the loop index;
- lack of skills in dealing with extremely complex nestings;
- integer division problem;
- poor initialization of a variable or pointer;
- source code that does not trace back to design;
- confusion regarding an alias for global data (global variable passed on to a subprogram).

1.3.6 Non-Compliance with Current Processes/Procedures

Some organizations have their own internal methodology and internal standards for developing/acquiring software. This internal methodology describes processes, procedures, steps, deliverables, templates, and standards (e.g., coding standard) that must be considered for software acquisition, development, maintenance, and operations. Of course, in a less mature organization, these processes/procedures will not be clearly defined.

We can therefore ask ourselves the following question: How can not fulfilling the requirements related to an internal methodology lead to defects in software? We must think in terms of the total life cycle (e.g., over many decades for subways and commercial airplanes) of the software, and not just of its initial development. It is clear that someone who only programs code appears to be far more productive than someone who develops intermediary products, such as requirements, test plans, and user documentation, as prescribed by the internal methodology of an organization. However, the immediate productivity would be disadvantageous in the long run.

Undocumented software will give rise to the following problems sooner or later:

- When members of the software team need to coordinate their work, they will have difficulty understanding and testing poorly documented or undocumented software.
- The person who replaces or maintains the software will only have the source code as a reference.

- SQA will find a large number of non-conformities (with respect to the internal methodology) regarding this software.
- The test team will have problems developing test plans and scenarios, primarily because the specifications are not available.

1.3.7 Inadequate Reviews and Tests

The purpose of software reviews and tests is to identify and check that errors and defects have been eliminated from the software. If these activities are not effective, the software delivered to the client will likely be prone to failure.

All kinds of issues can crop up when reviewing and testing software:

- reviews only cover a very small part of the software's intermediate deliverables;
- reviews do not identify all errors found in the documentation and software code;
- the list of recommendations stemming from reviews is not implemented or followed up on adequately;
- incomplete test plans do not adequately cover the entire set of functions of the software, leaving parts untested;
- the project plan has not left much time to perform reviews or tests. In some cases, this step is shortened because it is wedged between coding and the final delivery. Delays in the early steps of the project do not always mean the delivery date will be extended, to the detriment of proper testing;
- the testing process does not correctly report the errors or defects found;
- the defects found are corrected, but are not subject to adequate regression testing (i.e., retesting the complete corrected software) thereafter.

1.3.8 Documentation Errors

It has been recognized that obsolete or incomplete documentation for software being used in an organization is a common problem. Few development teams enjoy spending time preparing and reviewing documentation.

We would be inclined to say no to the question "does software wear out?" Indeed, the 0s and 1s found in the memories do not wear out from use as with hardware. In addition to classifying types of errors, it is important to understand the typical reliability curve for software. Figure 1.5 describes the reliability curve for computer hardware as a function of time. This curve is called a U-shaped or bathtub curve. It represents the reliability of a piece of equipment, such as a car, throughout its life cycle.

With regard to software, the reliability curve resembles more of what is shown in Figure 1.6. This means that software deterioration occurs over time due to, among other things, numerous changes in requirements.

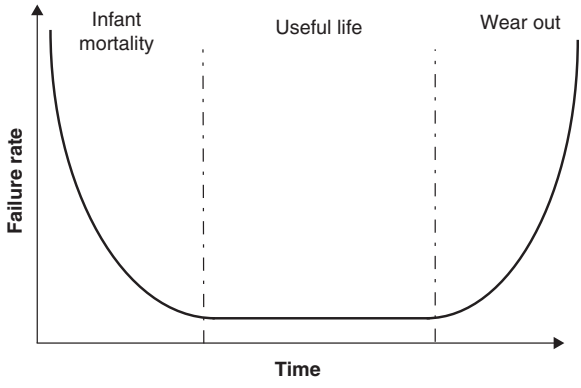


Figure 1.5 Reliability curve for hardware as a function of time.
Source: Adapted from Pressman 2014. [PRE 14].

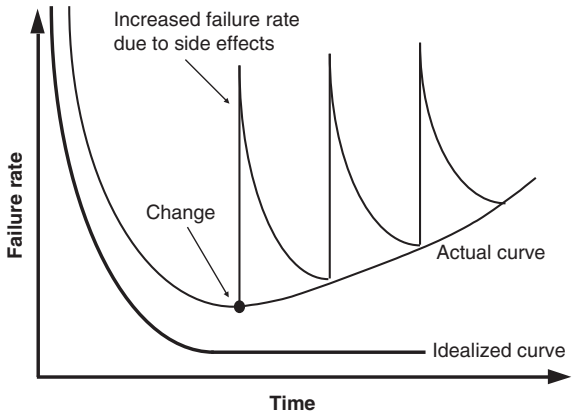


Figure 1.6 Reliability curve of software.
Source: Adapted from Pressman 2014. [PRE 14].

Professor April worked in the Middle East between 1998 and 2003 in a large telecommunications company. When he arrived, he noticed that the original documentation for critical application software for the telephone company had not been updated in over 10 years. There was no software quality assurance function in the information technology division of that company.

In conclusion, we see that there are many sources of potential errors, and that without SQA, these defects may result in failures if not discovered.

1.4 SOFTWARE QUALITY

The previous section, which presented the issues with identifying defects, has laid the ground work for our next discussion, namely software quality. How do we define software quality? The standards groups suggest the following definition.

Software Quality

Conformance to established software requirements; the capability of a software product to satisfy stated and implied needs when used under specified conditions (ISO 25010 [ISO 11i]).

The degree to which a software product meets established requirements; however, quality depends upon the degree to which those established requirements accurately represent stakeholder needs, wants, and expectations [Institute of Electrical and Electronics Engineers (IEEE 730)] [IEE 14].

The second definition in the text box is very different, despite appearances. The first part of the definition comes from the perspective of Crosby who reassures the software engineer with its strictness. This perspective is: "If I deliver all that is specified in the requirements document, then I will have delivered quality software." However, the second part of this definition is from the quality perspective of Juran, which specifies that one must satisfy the client's needs, wants, and expectations that are not necessarily described in the requirements documentation!

These two points of view force the software engineer to establish the kind of agreement that must describe client's requirements and attempt to faithfully reflect his needs, wants, and expectations. Of course, there is a practical element to the functional characteristics that need to be described, but also implicit characteristics, which are expected of any professionally developed piece of software.

In this context, the software engineer can be inspired by the standards in his field, just as his colleagues in construction engineering or other engineering specialties, in order to identify his obligations. Process conformance can be achieved and measured. As an example, Professor April published an example of the measurement, in Ouanouki and April (2007) [OUA 07], where the software testing process had to be assessed for Sarbanes-Oxley conformance for the largest Canadian hardware retailer.

Software quality is recognized differently depending on each perspective, including that of the clients, maintainers, and users. Sometimes, it is necessary to differentiate between the client, who is responsible for acquiring the software, and the users, who will ultimately use it.

Users seek, among other things, functionalities, performance, efficiency, accurate results, reliability, and usability. Clients typically focus more on costs and deadlines, with a view to the best solution at the best price. This can be considered an external point of view with regard to quality. To draw a parallel with the automobile

industry, the user (driver) will go to the garage that provides him with fast service, quality, and a good price. He has a non-technical point of view.

As for software specialists, they focus more on meeting obligations based on the allocated budget. Therefore, they see their obligations from the point of view of meeting requirements and the terms and conditions of the agreement. The choice of the right tools and modern techniques are often at the heart of concerns, and is therefore an internal point of view like that of a mechanic who is interested in the engine technology and knows it in detail. To him or her, quality is equally important with regard to the choice and assembly of components. We will consider these two points of view (external versus internal) when discussing the software product quality models.

Therefore, quality software is software that meets the true needs of the stakeholders while respecting any predefined cost and time constraints.

The client's need for software (or more generally any kind of system) may be defined at four levels:

– True needs
– Expressed needs
– Specified needs
– Achieved needs

The ability of software to meet (or not meet) the needs of the client can be described in the differences between these four levels. Throughout the development of a project, there will be factors that will affect the final quality.

For each level, Table 1.1 describes the typical factors that can affect the satisfaction of the client requirements.

1.5 SOFTWARE QUALITY ASSURANCE

This section presents a definition of SQA. This section also aims to describe the objectives of SQA. In order to put these definitions into perspective, here is a reminder of the general definition of software engineering:

Software Engineering

The systematic application of scientific and technological knowledge, methods, and experience for the design, implementation, testing, and documentation of software.

ISO 24765 [ISO 17a]

To be a recognized profession, software engineering must have its own body of knowledge for which there is consensus. As with most other engineering fields, recognized knowledge, methods, and standards must be used for the development, maintenance/evolution, and infrastructure/operation of software. The body of

Table 1.1 Factors that can Affect Meeting the True Requirements of the Client [CEG 90] (© 1990 - ALSTOM Transport SA)

Type of requirement	Origin of the expression	Main causes of difference
True	Mind of the stakeholders	– Unfamiliarity with true requirements – Instability of requirements – Different viewpoints of ordering party and users
Expressed	User requirements	– Incomplete specification – Lack of standards – Inadequate or difficult communication with the ordering party – Insufficient quality control
Specified	Software Specification Document	– Inappropriate use of management and production methods, techniques, and tools
Achieved	Documents and Product Code	– Insufficient tests – Insufficient quality control techniques

knowledge for software engineering is published in the SWEBOK guide (www.swebok.org). An entire chapter is dedicated to SQA.

Quality Assurance

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.

ISO 24765 [ISO 17a]

Software Quality Assurance

A set of activities that define and assess the adequacy of software processes to provide evidence that establishes confidence that the software processes are appropriate for and produce software products of suitable quality for their intended purposes. 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.

IEEE 730 [IEE 14]

The term "software quality assurance" could be a bit misleading. The implementation of software engineering practices can only "assure" the quality of a project, since the term "assurance" refers to "grounds for justified confidence that a claim has been or will be achieved." In fact, QA is implemented to reduce the risks of developing a software that does not meet the wants, needs, and expectations of stakeholders within budget and schedule.

This perspective of QA, in terms of software development, involves the following elements:

- the need to plan the quality aspects of a product or service;
- systematic activities that tell us, throughout the software life cycle, that certain corrections are required;
- the quality system is a complete system that must, in the context of quality management, allow for the setting up of a quality policy and continuous improvement;
- QA techniques that demonstrate the level of quality reached so as to instill confidence in users; and lastly,
- demonstrate that the quality requirements defined for the project, for the change or by the software department have been met.

In addition to software development, SQA can also focus on the maintenance/evolution and infrastructure/operations of software. A typical quality system should include all software processes from the most general (such as governance) to the most technical (e.g., data replication). QA is described in standards such as ISO 12207 [ISO 17], IEEE 730 [IEE 14], ISO 9001 [ISO 15], and exemplary practices models, such as CobiT [COB 12] and the Capability Maturity Model Integration (CMMI) models that will be presented in a later chapter.

1.6 BUSINESS MODELS AND THE CHOICE OF SOFTWARE ENGINEERING PRACTICES

In this section, Iberle (2002) [IBE 02], a senior test engineer at Hewlett-Packard, describes her experience in two business sectors of the same company: cardiology products and printers. Different business models are then described to help us understand the risks and the respective needs of each business sector with regards to software practices. These business models will be used in the following chapters to help choose or adapt software practices according to the context of a specific project or application domain.

Business Model

A business model describes the rationale of how an organization creates, delivers, and captures value (economic, social, or other forms of value). The essence of a business model is that it defines the manner by which the business enterprise delivers value to customers, entices customers to pay for value, and converts those payments to profit.

Adapted from Wikipedia

Knowledge of the business models and organizational culture will help the reader to [IBE 02]:

- evaluate the effectiveness of new practices for an organization or specific project;
- learn software practices from other fields or cultures;
- understand the context that promotes collaboration with members of other cultures;
- more easily integrate into a new job within another culture.

This section concludes with a brief discussion of exemplary software practices.

1.6.1 Description of the Context

Medical products belong to a field known for its very high quality standards. During a mandate in the cardiology products sector, Iberle (2002) [IBE 02] used a large number of traditional practices described in software engineering manuals, for example: detailed written specifications, intensive use of inspections and reviews throughout the life cycle, and exhaustive tests for requirements. Exit criteria were created at the beginning of the project and a product could not be shipped as long as the exit criteria were not met.

In this field, a project end date can be missed by weeks and even months. These delays are acceptable in order to fix any last-minute problems using a long checklist. It was far from painless. Iberle (2002) [IBE 02] explains that she worked many extra hours to try to be on schedule (and not exceed the deadline too much). There were heated debates as to whether a specific defect should be qualified as severe (level 1 severity) or average (level 2–5 severity). However, in the end, quality always won out over the schedule.

After 8 years of working on medical products, Iberle (2002) [IBE 02] was assigned to the business sector that produced printers and served small businesses and consumers. Practices in this business sector of the company were very different.

For example, specifications were far shorter, project exit criteria significantly less formal, but making the delivery date was very important. While Iberle was working in testing, she noticed differences in test practices. The main test effort was not focused on tests related to specifications. They were not trying to test all possible entry combinations. There was far less test documentation. In fact, some testers had no test procedures. This was a huge culture shock. At first, Iberle would walk around shaking her head, and grumbling "These people don't care about quality!" After a while, she started to see that her definition of quality was different and was based on her experience in a different field. It was time for her to revisit her beliefs about software quality.

1.6.2 Anxiety and Fear

When Iberle (2003) [IBE 03] worked on defibrillators and cardiographs, missing a delivery date was not the worst thing that could happen. What really scared the team was what could kill a patient or technician due to an electrical shock, cause a person to come to the wrong diagnosis, or that the device could not be used in an emergency situation. If the team raised the possibility of a failure, the delivery date was automatically pushed forward, without any discussion whatsoever. Lengthy and costly efforts to find and definitely eliminate the cause of the defect were systematically approved. It was obvious that, for an organization in this business sector, shirking one's legal responsibility or being blamed by the American Food and Drug Administration definitely contributed to these decisions. Delivery dates could be changed and production completed with overtime.

In the consumer products division, the reality was quite different. The potential for injury was very low, even in the worst conditions imaginable. The real concern was not respecting schedules or exceeding costs. When software has to be packaged in hundreds of thousands of boxes and these boxes must be sent to resellers on time for the day of a major sale, there is not much room to "play catch up." Another fear was having thousands of users unable to install their new printer and calling customer support lines the day after Christmas. Incompatibility between the most popular software and hardware was another source of concern.

So these two business divisions had different definitions of "quality." Clients valued different things: clients from the medical sector favored accuracy and reliability above all, whereas printer customers looked for user-friendliness and compatibility far more than reliability. Of course, everyone wants reliability. However, whether they are aware of it or not, people value reliability as a function of the pain that certain problems may cause them. People are not happy when they have to restart their computer from time to time, but their misfortune is nothing in comparison with the anguish of a patient faced with a functional problem with a heart defibrillator. When someone goes into fibrillation, there is a 5–6 minute window for saving the patient. So there is no time to lose with equipment problems.

The definition of "reliability" is therefore also very different in these two business sectors. When it was understood that no one would die from a printer software error, the team examined the software practices in the medical products division to determine whether they were also useful in the printer sector [IBE 03]. It would take Iberle several months to realize that what seemed shoddy in the printer sector was a way of dealing with different priorities that did not carry the same weight as for medical products.

1.6.3 Choice of Software Practices

As expected, people from both business sectors chose software engineering practices that would lower the probability of their worst fears. Since their apprehensions are different, their practices are also different. In fact, in light of their fears, the choice of practices starts to make sense. The fear of a false diagnosis leads to many detailed reviews and various types of tests. However, the fear of confusing printer users results in more usability tests.

It is not surprising to see that people who work in the same business sectors have similar concerns and use similar practices. Certain concerns can also be found in other organizations. For example, the aerospace sector and medical sector are very closely related. It is also possible for the same organization to have different fears and values in different business sectors, as Iberle (2003) [IBE 03] described above of her employment at Hewlett-Packard.

Software organizations or software specialists are divided into groups that appreciate similar things or share the same concerns, based on similarities in client and business community expectations. These cultures are called "practice groups," that is, software development groups, which share common definitions of quality and tend to use similar practices.

1.6.4 Business Model Descriptions

The following models were developed by Iberle to better understand the need for QA in different business sectors, given that the way in which money flows through an organization (e.g., contract income, cost of products delivered, and losses) and how profits are generated affect the choice of the software practices used to develop products for an organization. The five main business models in the software industry are [IBE 03]:

- Custom systems written on contract: The organization makes profits by selling tailored software development services for clients (e.g., Accenture, TATA, and Infosys).
- Custom software written in-house: The organization develops software to improve organizational efficiency (e.g., your current internal IT organization).

- Commercial software: The company makes profits by developing and selling software to other organizations (e.g., Oracle and SAP).
- Mass-market software: The company makes profits by developing and selling software to consumers (e.g., Microsoft and Adobe).
- Commercial and mass-market firmware: The company makes profits by selling software in embedded hardware and systems (e.g., digital cameras, automobile braking system, and airplane engines).

1.6.5 Description of Generic Situational Factors

Each business model has a set of attributes or factors that are specific to it. Here is a list of situational factors that seem to influence the choice of software engineering practices in general [IBE 03]:

- Criticality: The potential to cause harm to the user or prejudice the interests of the purchaser varies depending on the type of product. Some software may kill a person if it shuts down; other software programs may result in major money losses for many people; others will make a user waste time.
- Uncertainty of users' wants and needs: The requirements for software that implements a familiar process in an organization are better known than the requirements for a consumer product that is so new that the end-users do not even know what they want.
- Range of environments: Software written for use in a specific organization only has to be compatible with its own computer environment, whereas software sold to a mass market must work in a wide range of environments.
- Cost of fixing errors: Distributing corrections for certain software applications (e.g., embedded software of an automobile) is usually far more costly than fixing a website.
- Regulations: Regulatory bodies and contractual clauses may require the use of software practices other than those that would normally be adopted. Certain situations require process audits to check whether a process was followed at the time of producing the software.
- Project size: Projects that take several years and require hundreds of developers are common in certain organizations, whereas in other organizations, shorter projects developed by a single team are more typical.
- Communication: There are a certain number of factors, in addition to project scope, that can increase the quantity of person-to-person communication or make communications more difficult. Certain factors seem to occur more often within certain cultures, whereas others happen at random:
  - Concurrent developer–developer communication: Communication with other people on the same project is affected by the way in which the work is

distributed. In certain organizations, senior engineers design the software and junior staff carries out the coding and unit tests (instead of having the same person carrying out the design, coding, and unit tests for a given component). This practice increases the quantity of communications between developers.
    - Developer–maintainer communication: Maintenance and enhancements require communication with the developers. Communication with developers is greatly facilitated when they work in the same area.
    - Communication between managers and developers: Progress reports must be sent to upper management. However, the quantity of information and form of communication that managers believe they need may vary substantially.
- Organization's culture: The organization has a culture that defines how people work. There are four types of organizational cultures:
    - Control culture: control cultures, such as IBM and GE, are motivated by the need for power and security.
    - Skill culture: A culture of skill is defined by the need to make full use of one's skills: Microsoft is a good example.
    - Collaborative culture: A collaborative culture, as illustrated by Hewlett-Packard, is motivated by a need to belong.
    - Thriving culture: A thriving culture is motivated by self-actualization, and can be seen in start-up organizations.

1.6.6 Detailed Description of Each Business Model

This section goes into more detail about each of the five main business models. A single business model, contract-based development for made-to-measure systems, is described as an in-depth case study. For this business model, we describe the following four perspectives:

- context;
- situational factors;
- concerns; and
- software practices predominately used in this business model.

For the other four business models, we will only consider the context and concerns.

1.6.6.1 Custom Systems Written on Contract

In a fixed-price contract, Iberle (2003) [IBE 03] indicated that the client specifies exactly what he wants and promises the supplier a given sum of money. The profits made by the supplier depend on his ability to remain within budget and to deliver on schedule, as defined in the contract, a product that performs as intended. Large-scale

applications and military software are often written under contract. The software produced in this business culture is often critical software. The cost of distributing fixes after delivery is manageable because the corrections are provided to an environment that is known and accessible, and to a reasonable number of sites.

Critical Software

Software having the potential for serious impact on the users or environment due to factors including safety, performance, and security.

Adapted from ISO 29110 [ISO 16f]

Critical System

System having the potential for serious impact on the users or environment due to factors including safety, performance, and security.

ISO 29110 [ISO 16f]

Following is the list of dominating factors in this business model [IBE 03]:

- Criticality: Software failures in financial systems can seriously compromise the client's business interests. Software defects in planes and military systems may endanger lives, even if many software programs purchased by the Defense Department are business software applications whose failure would have the same impact as that in financial systems.
- Uncertainty of user needs and requirements: Since buyers and users are an identifiable group, they can be contacted to find out what they are looking for. In general, they have a relatively clear idea of what they want. However, the process to put this into place is not always well documented, and users may not agree on the steps in the process, their demands may require technology that does not exist, business needs may change during the project, and sometimes people completely change their minds.
- Range of environments: In general, the purchasing organization has identified a small set of target environments in order to avoid cost increases. The result is a range of environments that are clearly defined and relatively small, compared with other cultures.
- Cost of fixing errors: In general, there are inexpensive ways of distributing fixes—a large portion of the software will be on servers in a given building and the client's software location is generally known.
- Regulations: Defense software (e.g., for a fighter or commercial plane) must comply with a huge list of regulations, most of which concern the software development process. Financial software is not subject to regulations in the

same way. It is common that the contract will stipulate process audits to prove that the organization followed its development process. The client expects to receive regular progress reports on the project.

- Project size: Often large or even immense. Several dozen people work for more than 2 years on the average-size project, but hundreds of people over several years are required for large projects. There is also some data that indicates that small projects are far more common than large projects.
- Communication: The practice involving dividing architecture and coding between senior and junior professionals is occasionally observed in this culture. Given that the systems and projects are large, often different people and even separate departments are used for analysis, design, etc. Moreover, maintenance contracts can go to people other than the original developers. This may produce competition and make communications more complex. Organizations that develop software are often large, whether their projects are small or large, which means additional hierarchical levels.
- Organizational culture: Organizations that write software on contract often have a control culture. This seems logical since most of them have ties with the military.

The concerns of the developers of these systems are often:

- incorrect results;
- exceeding budget;
- penalties for late delivery, and
- not delivering what the client asked for (which may lead to legal proceedings).

These situational factors lead to certain assumptions regarding this business model:

- delivery on schedule and within budget is imperative;
- reliable, correct software is imperative;
- requirements must be known and detailed from the project onset;
- projects are typically large scale with many communication channels;
- it is necessary to show that what was promised has indeed been delivered;
- plans must be developed, and regular progress reports prepared (which are sent to project management and the client).

In the text above, we presented three perspectives: context; situational factors; and concerns about the first business model.

In the next few paragraphs, we present the prevailing practices used with the business model of this case study.

These practices are taken from [IBE 03]:

– A lot of documentation

  Documentation is a valuable way of communicating when the project size is large and when external suppliers are involved. Written documentation is often far more effective than discussions around the cooler when the communication channels are complex, which occurs when people are geographically remote and in different organizations. In addition, certain documents are often necessary to prove that we are doing what was set out in the contract. Lastly, in order for the requirements to be known in detail at the start of the project, documentation and many reviews of the requirements are necessary before responding to the call for tenders.

– Lists of exemplary practices

  Lists of exemplary practices, such as the CobiT [COB 12] and CMMI models, developed by the Software Engineering Institute, are used to develop contractual clauses. For example, in this business model, the focus is on project estimating and management in order to be on schedule and within budget as stipulated in the contract, and regular progress reports are necessary.

– Waterfall development cycle

  The waterfall development life cycle was invented in the 1950s to provide enough structure for large IT projects to be able to plan and strategize on-time delivery. The new iterative and agile development cycles plan out development in smaller increments, which allow for planning while offering more flexibility as to delivery. However, as it has been observed, in this business model, cascade development is often the preferred method.

– Project audits

  Audits are often specified in the contract for this business model. The audit is used to prove to the client, or during legal proceedings, that the contractual clauses, such as respecting schedules, quality, and functions, have been fulfilled.

We have now described the four perspectives for this business model: the context; situational factors; concerns; and predominant practices. In the next section, we present, as described by Iberle (2003) [IBE 03], only the context and concerns regarding the other four business models.

1.6.6.2 Custom Systems Written In-House

When using one's own employees to develop software, economic aspects are different than for those who have their software developed on a contract basis. The value of the work depends on improving efficacy or efficiency of operations within the organization. Less focus is put on scheduling meetings since projects are often pending or

postponed depending on the budget. The systems can be critical for the organization or of an experimental nature. Fixes are distributed to a limited number of sites.

Developers of these systems often are concerned with the following:

– producing incorrect results;
– limiting the ability of other employees to do their own work;
– their project being cancelled.

1.6.6.3 Commercial Software

Commercial software is software sold to other organizations rather than to an individual consumer. Profits depend on the familiar economic model, which involves selling many copies of the same piece of software for more than the cost of developing and making the copies. Instead of meeting the specific needs of a single client, the developer aims to satisfy many clients. The software is often critical for the organization or at least very important for the client's organizational operations. Since the software is in the hands of many clients in many places, the distribution of corrections can be very costly. These clients also tend to instigate legal proceedings if the software is deficient, which increases the cost of errors.

Business system vendors are generally fearful of:

– court cases;
– recalls;
– tarnishing their reputation.

1.6.6.4 Mass-Market Software

This software is sold to individual consumers often at a very high volume. Profits are made by selling products at higher than development cost, often in a niche market or at certain times of the year, such as at Christmas. The potential effects of software failures for the client are generally less serious than those in the previous models and clients are less likely to demand reparation for any damage incurred. The failure of certain software may considerably affect the user's well-being, such as in the case of tax preparation software. However, for most, a failure is simply a source of frustration.

The typical concerns in this culture are:

– missing the marketing opportunity;
– a high level of support calls;
– bad reviews in the press.

The cost of fixing errors, for mass-market product manufacturers, could be significantly reduced when the owner can update their products. Unfortunately, the customer will be left to search for and perform these upgrades.

1.6.6.5 Commercial and Mass-Market Firmware

Given that profits depend on the sale of the product for more than the manufacturing cost, the cost of distributing fixes is extremely high, since electronic circuits must often be changed on site. Corrections cannot simply be sent to the client. The impact of down time with mass-market embedded software is potentially more serious than the impact of software failures, since the software is controlling a device. Although the destructive potential of small objects, such as digital watches, is low, in certain cases, software failures could have fatal consequences.

The typical concerns of this culture are:

– incorrect behavior of the software in certain situations;
– recalls;
– court cases.

Business Model—Open-Source Software

Open-source software is software that is distributed with its source code and the authorization to modify and distribute it freely under the condition that it is also provided as open-source software once modified.

This business model is becoming an influential economic model in the software industry. It permits its users to collaborate, essentially over the Internet, by adding improvements to a software and distributing it once modified. This approach allows others to benefit from these innovations. However, open-source software does not permit its developers to be paid for these improvements.

The concerns associated with this model are:

– undemonstrated quality;
– lack of support;
– delays in providing fixes.

Adapted from Wikipedia

1.7 SUCCESS FACTORS

Implementing practices to improve software quality can be facilitated or slowed down based on factors inherent to the organization. The following text boxes list some of these factors.

Factors that Foster Software Quality

1) SQA techniques adapted to the environment.
2) Clear terminology with regards to software problems.
3) An understanding and specific attention to each major category of software error sources.
4) An awareness of the SQA body of knowledge of the SWEBOK as a guide for SQA.

Factors that may Adversely Affect Software Quality

1) A lack of cohesion between SQA techniques and environmental factors in your organization.
2) Confusing terminology used to describe software problems.
3) A lack of understanding or interest for collecting information on software error sources.
4) Poor understanding of software quality fundamentals.
5) Ignorance or non-adherence with published SQA techniques.

1.8 FURTHER READING

ARTHUR L. J. *Improving Software Quality: An Insider's Guide to TQM*. John Wiley & Sons, New York, 1992, 320 p.

CROSBY P. B. *Quality Is Free*. McGraw-Hill, New York, 1979, 309 p.

DEMING W. E. *Out of the Crisis*. MIT Press, Cambridge, MA, 2000, 524 p.

HUMPHREY W. S. *Managing the Software Process*. Addison-Wesley, Reading, MA, 1989, Chapters 8, 10, and 16.

JURAN J. M. *Juran on Leadership for Quality*. The Free Press, New York, 1989.

SURYN W., ABRAN A., and APRIL A. ISO/IEC SQuaRE. The Second Generation of Standards for Software Product Quality. In: Proceedings of the 7th IASTED international conference on Software Engineering and Applications (ICSEA'03), Montreal, Canada, 2003, pp. 1–9.

VINCENTI W. G. *What Engineers Know and How They Know It—Analytical Studies from Aeronautical History*. John Hopkins University Press, Baltimore, MD, 1993, 336 p.

1.9 EXERCISES

1.1 Describe the difference between a defect, an error, and a failure.

1.2 According to the studies of Boris Beizer, when do the greatest number of software errors occur in the software development life cycle?

1.3 Describe the difference between the software and hardware reliability curves.

1.4 Eight categories for causes of errors describe the development and maintenance environment, as experienced in organizations:

a) Identify and describe these situations.

b) What situations more specifically influence software engineers who develop and maintain the software?

c) What situations more specifically influence the effort of the software engineering managers who develop and maintain the software?

1.5 Describe the different perspectives of software quality from the point of view of the client, the user, and the software engineer.

1.6 Describe the types of needs, their origin, and the causes for differences that may be due to a discrepancy between the needs expressed by the client and those carried out by the software engineer.

1.7 Describe the concept of business models and how it creates different perspectives for SQA requirements.

1.8 Describe the main differences between QA and quality control.