Course Material
Module 1: Essential SQA: Processes and Success Factors

Topic No.	Topic Title	Reference
1.1	Definition and importance of software quality assurance	T1 Chapter 1
1.2	Distinction between Quality Assurance and Quality Control	Lecture Notes
1.3	Success Factors in Quality Assurance	T1 Chapter 1 & T2 Chapter 1
1.4	Cost of Quality and Quality Culture	T1 Chapter 2
1.5	Role of SQA in software development life cycle	Lecture Notes

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 quaity 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 com plex 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 docu mentationand data pertaining to the operation of a computer system

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 de fined 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

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.

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 a defect in the test plan.
- I found 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.

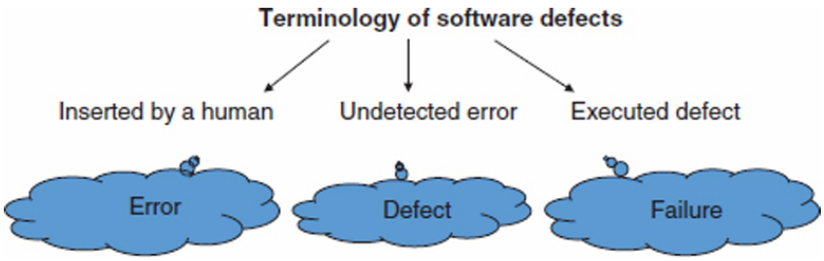


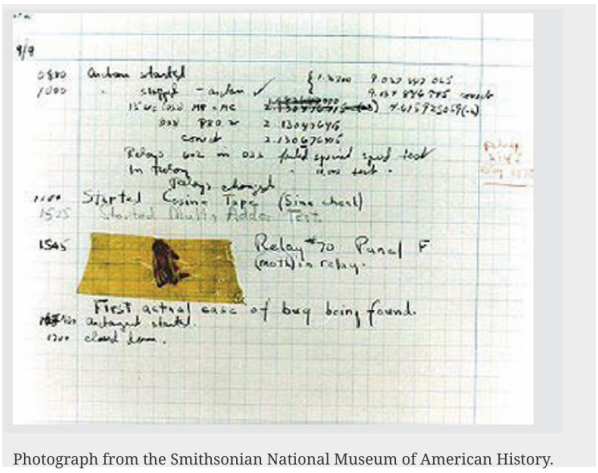
Figure 1.2 Terminology recommended for describing software problems.

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 ap plied to computers and computer programs, started to appear in the popular press [KID 98]



Photograph from the Smithsonian National Museum of American History.

Falure

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 normal.

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

1. 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 require ments specifications and design) and the source code itself **Failures occur when an intermediary product or faulty software is used.**

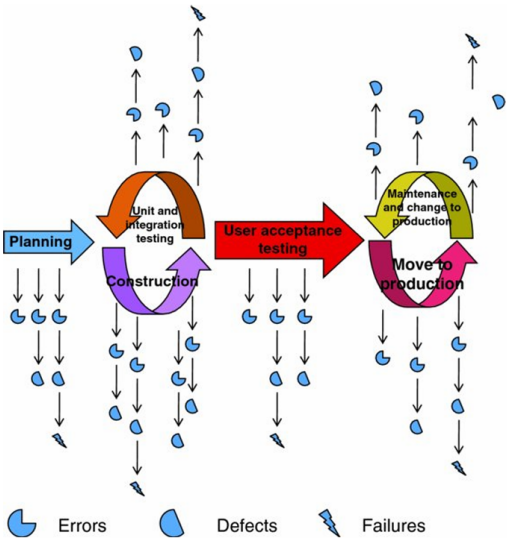


Figure 1.3 Errors, defects, and failures in the software life cycle.

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 sys tem (due to a poor understanding of the algorithm to set up for this requirement) involved only taking into account the average amount of current purchases and not the customer's monthly his tory. 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 in formed 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.

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

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.

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.

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**.

- 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 have explored error rates in software development, highlighting the impact of software engineering maturity and developer competency. McConnell (2004) noted that fewer errors occur with mature processes. Humphrey (2008) observed developers unintentionally create about 100 defects per 1000 lines of code, with variations from 50 to over 250 defects among experienced developers. Rolls-Royce reported even lower rates, ranging from 0.5 to 18 defects per 1000 lines. Proven processes, skilled developers, and reusing tested components significantly reduce software errors.

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 program ming 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

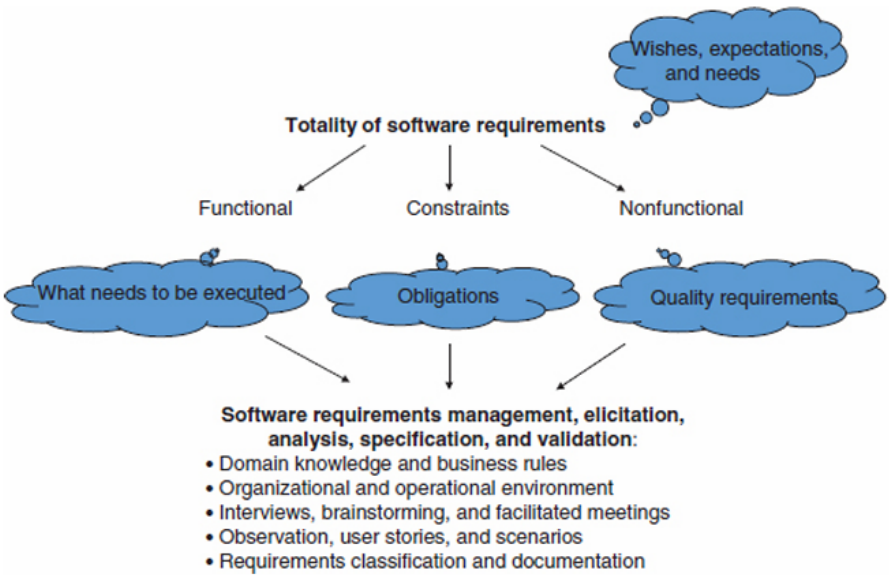
1.3.1 Problems with Defining Requirements
Defining software requirements is now considered a specialty, which means a business analyst or a software engineer specialized in requirements.

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
- **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 dif icult to cater to the wishes, expectations, and needs of many different user groups at the same time (see Figure 1.4).



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

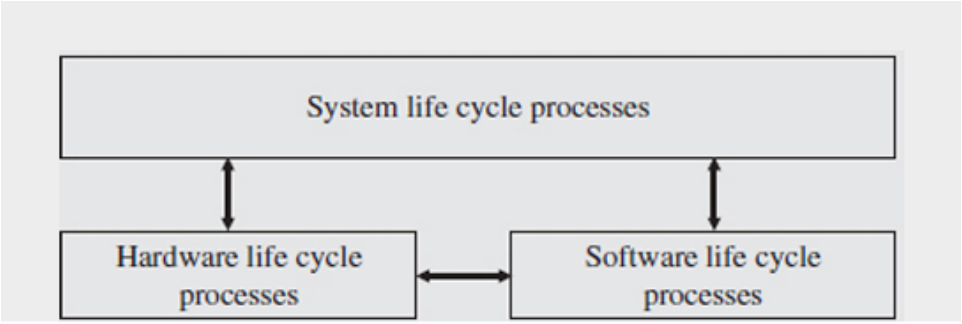
- Correct
- Complete
- clear for each stakeholder group (e.g., the client, the system architect, testers, and those who will maintain the system)
- unambiguous, that is, same interpretation of the requirement from all stakeholders
- concise (simple, precise)
- consistent
- feasible (realistic, possible)
- 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

Ambler [AMB 04] critiques the traditional "Big Requirements Up Front" approach, calling it outdated due to its inefficiency and risk of project failure. He highlights that many specifications are unused and documentation often remains outdated. Instead, he advocates for agile methods like Test-Driven Development to minimize documentation.

Prototyping is often used by software analysts and designers to replace traditional requirement documents with user interfaces and test cases, clarifying requirements and design while gathering early client feedback.

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 figure illustrates the interactions between their life cycle processes.

Systems engineering collaborates closely with hardware and software engineering to allocate system requirements, including functionalities and quality aspects like safety and performance.



1.3.2 Maintaining Effective Communications Between Client and Developer

Errors often stem from misunderstandings between software teams and clients. **Developers must communicate in clear, non-technical language, consider the user's perspective, and remain attentive to signs of miscommunication on both sides.**

They must be aware of all signs of lack of commu nication, 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)

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 occur during software construction. McConnell (2004) [MCC 04], in his book Code Complete, outlines effective techniques for writing quality source code and highlights common programming errors.

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 nesting
- 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 establish internal methodologies and standards for software development, encompassing processes, procedures, deliverables, and coding standards.

In less mature organizations, these may not be well-defined. Non-compliance with internal methodologies can lead to software defects, especially when considering the software's full life cycle, spanning decades for systems like subways and airplanes.

While focusing solely on coding might seem more productive, neglecting intermediary deliverables—such as requirements, test plans, and documentation—results in long-term disadvantages.

Undocumented software eventually leads to the following issues:

- Software team members will struggle to understand and test poorly documented or undocumented software, hindering coordination.

- Replacement or maintenance personnel will rely solely on the source code, lacking proper guidance.
- QA teams will identify numerous non-conformities with internal methodologies.
- Testing teams will face difficulties developing test plans and scenarios due to the absence of specifications.

1.3.7 Inadequate Reviews and Tests

Software reviews and tests ensure errors are eliminated; ineffective testing risks client software failure**.**

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

- Reviews cover only a small portion of the software's intermediate deliverables.
- Reviews fail to identify all errors in documentation and software code.
- Recommendations from reviews are not adequately implemented or followed up.
- Incomplete test plans leave parts of the software untested.
- Limited time for reviews and tests due to delays in earlier project stages.
- Testing processes do not report errors or defects accurately.
- Corrected defects are not subjected to sufficient regression testing.

1.3.8 Documentation Errors

Obsolete or incomplete software documentation is a common issue as development teams often neglect it. Unlike hardware, software doesn't wear out physically, but its reliability deteriorates over time due to frequent changes in requirements.

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

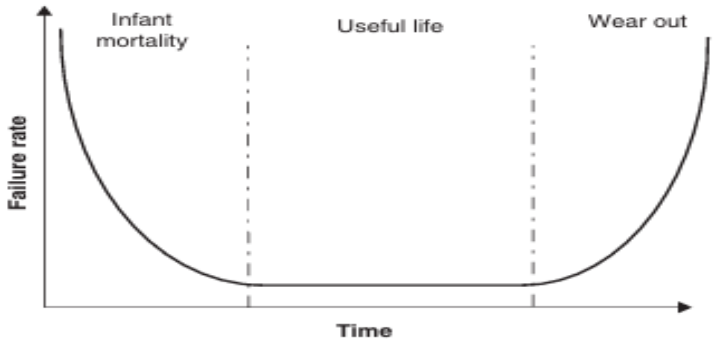


Figure 1.5 Reliability curve for hardware as a function of time.
Source: Adapted from Pressman 2014. [PRE 14].

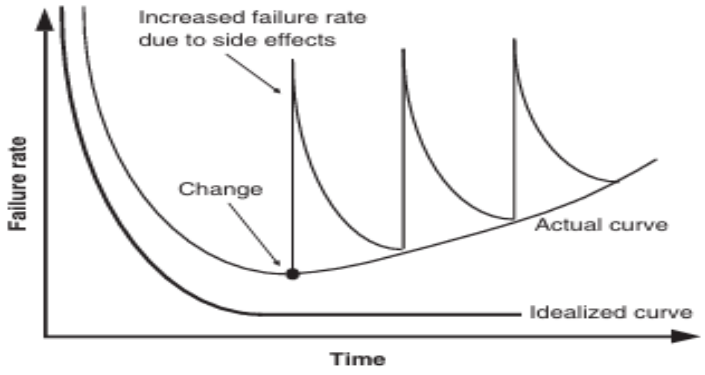


Figure 1.6 Reliability curve of software.
Source: Adapted from Pressman 2014. [PRE 14].

1.4 SOFTWARE QUALITY

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 rep resent stakeholder needs, wants, and expectations [Institute of Electrical and Electronics Engineers (IEEE 730)] [IEE 14]

Crosby defines quality as delivering all specified requirements, while Juran emphasizes meeting the client's implicit needs and expectations beyond documentation.
Software quality is recognized differently depending one ach 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.

These perspectives require software engineers to align requirements with both functional and implicit client expectations

Quality varies by perspective:

- **Users** value functionality, performance, reliability, and usability.
- **Clients** prioritize costs and deadlines for optimal solutions.

Software specialists prioritize meeting obligations within the allocated budget, focusing on fulfilling requirements and agreement terms. Their primary concern lies in selecting the right tools and modern techniques, adopting an internal perspective akin to a mechanic who values engine technology and component quality. These internal and external viewpoints will be explored in the context of software product quality models.

Therefore, quality software is software that meets the true needs of the stake holders 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 need

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

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 – Insufficient tests
Achieved	Documents and Product Code	– Insufficient quality control techniques

1.5 SOFTWARE QUALITY ASSURANCE

Quality Assurance

1. A planned and systematic pattern of 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. Planned and systematic activities implemented within the quality system, demonstrated as needed, to provide adequate confidence that an entity will fulfil quality requirements.

Software Quality Assurance

Software Quality Assurance (SQA) involves activities that ensure software processes are effective and produce quality products for their intended purpose.
A key aspect of SQA is its objectivity and independence from project pressures, ensuring unbiased evaluation.

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 throughout the software life cycle to identify and implement necessary corrections.
- A quality system that enables the establishment of a quality policy and supports continuous improvement within quality management.
- QA techniques to demonstrate the achieved quality level and build user confidence.
- Evidence that the quality requirements defined for the project, changes, or by the software department have been met.

2. Distinction between Quality Assurance and Quality Control

Definitions

- **Quality Assurance (QA):** QA refers to the planned and systematic activities implemented to ensure that processes and procedures are adequate to meet quality standards. It focuses on preventing defects by improving the processes used to create the product.
- **Quality Control (QC):** QC involves the activities and techniques used to inspect and test the final product to ensure it meets the defined quality standards. It focuses on identifying and fixing defects in the product.

Quality Assurance (QA) refers to the planned and systematic activities implemented to ensure that processes meet established quality standards and prevent defects.

In contrast, Quality Control (QC) refers to the activities and techniques carried out to inspect and test the final product, ensuring it meets the defined quality standards.

Aspect	Quality Assurance (QA)	Quality Control (QC)

Definition	Planned and systematic activities to ensure processes meet quality standards.	Activities and techniques to inspect and test the final product.
Focus	Processes and procedures.	The product itself.
Objective	Ensures development/manufacturing processes are efficient and follow standards.	Ensures the product meets quality standards and specifications.
Techniques	Process standardization, audits, reviews, documentation, and training.	Testing, inspections, statistical sampling, and analysis.
Timing	Occurs throughout the development or production lifecycle.	Conducted after the product is developed or during production.
Role	Ensures the "right process" is followed to achieve quality; proactive and preventive.	Ensures the "right product" is delivered; reactive and corrective.
Perspective	Organizational responsibility involving management-level planning and control.	Team or project-level responsibility focused on the product.
Examples	Establishing coding standards, conducting process audits.	Performing unit testing, inspecting batches for defects.

1.3 Success Factors in Quality Assurance

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

Explanation:
The **SQA (Software Quality Assurance) body of knowledge** within the **SWEBOK (Software Engineering Body of Knowledge)** serves as a comprehensive guide for ensuring software quality throughout the development lifecycle. Here's a breakdown:

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

McCall's Factor Model

McCall's model classifies software requirements into **11 quality factors**, grouped into three categories. based on their relevance to the software's **operation, revision, and transition phases** These factors highlight different dimensions of quality to ensure software meets functional and non-functional requirements throughout its lifecycle.

1. Product Operation Factors

These factors ensure the software's performance during daily operation:

- **Correctness:**
  Refers to the software's **ability to provide accurate and complete results**.
  - **Example**: A sales system displaying correct **customer balances** or an industrial system maintaining **precise temperature control**.
- **Reliability:**
  Ensures the software performs **consistently without failure** under specified conditions.
  - **Example:** A banking application processing transactions without crashing.
- **Efficiency:**
  Measures the **optimal use of resources** like **processing time, memory, or power.**
  - **Example:** A mobile app that consumes minimal battery while running.
- **Integrity:**
  Ensures **data protection and prevents unauthorized** access or tampering.
  - **Example:** Secure login systems for sensitive applications.
- **Usability:**
  Focuses on ease of use and user satisfaction, including intuitive interfaces and help documentation.
  - **Example:** A user-friendly e-commerce platform.

2. Product Revision Factors

These factors address the software's ability to be updated, maintained, or tested:

- **Maintainability:**
  Refers to how easily developers can **identify and fix defects or make updates**.
  - **Example:** A codebase with clear documentation and modular design.
- **Flexibility:**
  The ability **to adapt to changing requirements or environments**.
  Example: A payroll system accommodating new tax regulations.
- **Testability:**
  Ensures that **software can be tested comprehensively and defects can be identified**.
  - **Example:** A system designed with automated test cases and logging.

3. Product Transition Factors

These factors relate to the **software's adaptability to new environments or systems**:

- **Portability:**
  The ease of **transferring software to different hardware or operating systems**.
  - **Example:** A web application running seamlessly on different browsers.
- **Reusability:**
  The capability to **repurpose components** in other applications.
  - **Example**: A **library of authentication** modules used **across multiple projects**.
- **Interoperability:**
  The ability to integrate and communicate effectively with **other systems**.
  - **Example:** A hospital management system exchanging data with external diagnostic tools.

Importance of McCall's Model

McCall's model provides a structured approach to ensuring comprehensive software quality by addressing:

1. **Daily operation:** Reliability, correctness, and usability.
2. **Future revisions:** Maintainability and flexibility.
3. **Adaptability:** Portability and interoperability.

By focusing on these aspects, organizations can deliver software that is robust, scalable, and aligned with both user and business needs.

1.4 Cost of Quality and Quality Culture

One of the major factors that explains the resistance to implementing quality assurance is the perception of its high cost. In organizations that develop software, it is rare to find data on the cost of non-quality.

This section defines the components of the cost of quality, explores its application in a major American military company, and presents data collected by Professor Claude Y. Laporte from various organizations..

Presenting SQA in terms of costs helps administrators understand its value and aligns it with their financial priorities. This approach **positions SQA as a business investment**, making it easier to gain support and ensuring it fits with the company's other initiatives.

Software engineers must inform administrators about the risks of neglecting software quality. A practical starting point is identifying the costs of non-quality, as analyzing software-related problems helps highlight potential savings and the benefits of quality improvements.

The costs of a project can be grouped into four categories:
(1) implementation costs,
(2) prevention costs,
(3) appraisal costs, and
(4) the costs associated with failures or anomalies

If all development activities were error-free, costs would solely cover **implementation**. However, mistakes require detection, leading to **appraisal costs** (e.g., testing). Errors that slip through result in **anomaly costs**. To reduce these, investments in training, tools, and methodologies are made, categorized as **prevention costs**.

The "cost of quality" is not calculated in the same way in all organizations.

The commonly used model for costs of quality includes five perspectives:

The commonly used model for costs of quality includes five perspectives:

1. **Prevention Costs:** Investments in training, tools, and methodologies to avoid errors.
2. **Appraisal Costs:** Costs of detecting errors, such as testing.
3. **Internal Failure Costs:** Costs of fixing errors during development.
4. **External Failure Costs:** Costs of addressing errors at the client's site.
5. **Warranty and Reputation Costs:** Costs from warranty claims and damage to reputation due to poor quality.

The calculation of the cost of quality in this model is as follows

$$\begin{aligned}\text{Quality costs} = {} & \text{Prevention costs} \\ & + \text{Appraisal or evaluation costs} \\ & + \text{Internal and external failure costs} \\ & + \text{Warranty claims and loss of reputation costs}\end{aligned}$$

In addition to quality-related costs, defective software can lead to severe repercussions, including:

- **Lawsuits and court-imposed penalties.**
- **Decline in market share value.**
- **Loss of financial partners.**
- **Departure of key employees.**

These impacts highlight the broader risks of poor software quality.

Krasner [KRA 98] published a table to better understand the activities and costs for three quality perspectives (see Table 2.1

Major Categories	Subcategories	Definition	Typical Costs
Prevention Cost	Quality Basis Definition	Effort to define quality, set goals, standards, and thresholds, Quality trade-off analysis,	Definition of release criteria for acceptance testing, and related quality standards.
	Project and Process-Oriented Interventions	Effort to prevent poor product quality or improve process quality.	Process improvement, updating procedures and work instructions, metric collection and analysis, internal and external quality audits, employee training, and certification of employees
Appraisal/Evaluation Cost	Discovery of Product Condition	Effort to discover the condition and level of non-conformance in the product.	Testing, walkthroughs, inspections, desk checks, and quality assurance activities.
	Ensuring Quality Achievement	Effort to ensure quality control before proceeding with production or delivery.	Contract/proposal reviews, product quality audits, “go/no-go” decisions, and subcontractor quality assurance.
Cost of Anomalies/Non-Conformance	Internal Anomalies/Non-Conformance	Problems detected before delivery to the customer.	Rework activities such as recoding, retesting, re-reviewing, and re-documenting.
	External Anomalies/Non-Conformance	Problems detected after delivery to the customer.	Warranty support, complaint resolution, reimbursements, damage payments, project delays, reduced sales, reputation damage, and increased marketing efforts.

Cost of quality models, introduced in the 1950s by Feigenbaum, classify quality assurance costs from an economic perspective.

Increased detection and prevention costs reduce failure costs (internal and external), while reducing detection and prevention efforts raises failure costs. High-criticality software requires higher quality levels, resulting in increased detection and prevention costs.

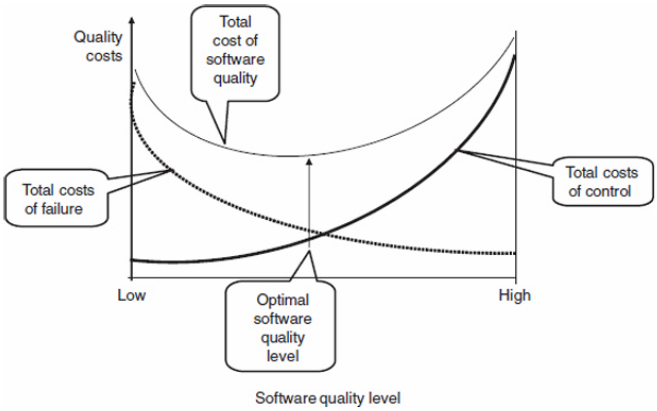


Figure 2.1 Balance between the software quality level and the cost of quality [GAL 17].

The primary goal of SQA is to demonstrate its benefits to management, emphasizing that early error detection saves time, money, and effort.

Capers Jones [JON 00] highlights that fixing defects becomes exponentially costlier as they propagate through development phases, with operational phase fixes costing up to 100 times more. Similarly, Boehm [BOE 01] noted that correcting a defect in the requirements phase costs $25, while fixing it during programming costs $139.

Early detection is essential to minimizing defect-related costs.

Figure 2.3, as presented by Selby and Selby (2007) [SEL 07], illustrates the distribution of defect injection across the software development life cycle. **It highlights the stages where defects are most commonly introduced**, emphasizing the importance of early detection and resolution to reduce overall costs and improve software quality.

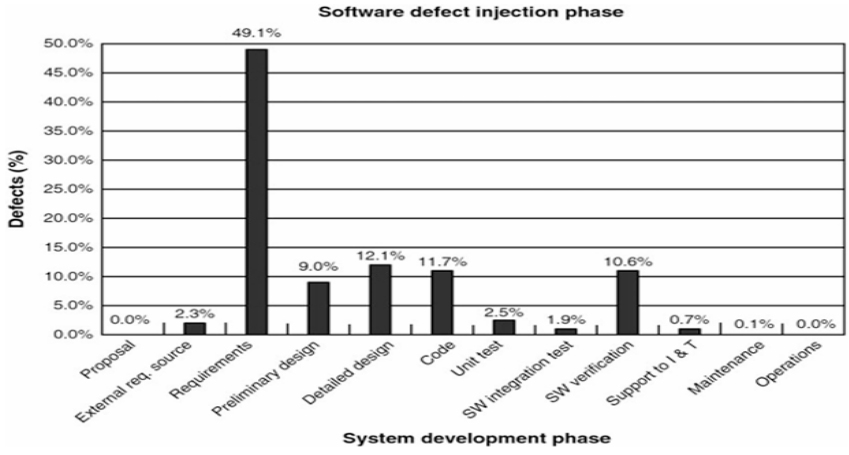


Figure 2.3 Defect injection distribution during the life cycle [SEL 07].

In this case study, about **50% of defects occur during the requirements phase**. If not fixed early, these defects become costly to address in later phases and may delay delivery. Meeting strict deadlines often results in releasing software with unresolved defects, leaving clients to use flawed versions until updates are provided.

Figure 2.4 illustrates A study by Raytheon in the defense industry measured the cost of quality over nine years.
Initially, rework costs were 41% of total project costs. These decreased to 18% with improved process maturity, 11% at level 3 maturity, and 6% at level 4 maturity. Prevention costs, when kept under 10% of total project costs, significantly reduced rework costs from 45% to below 10%.

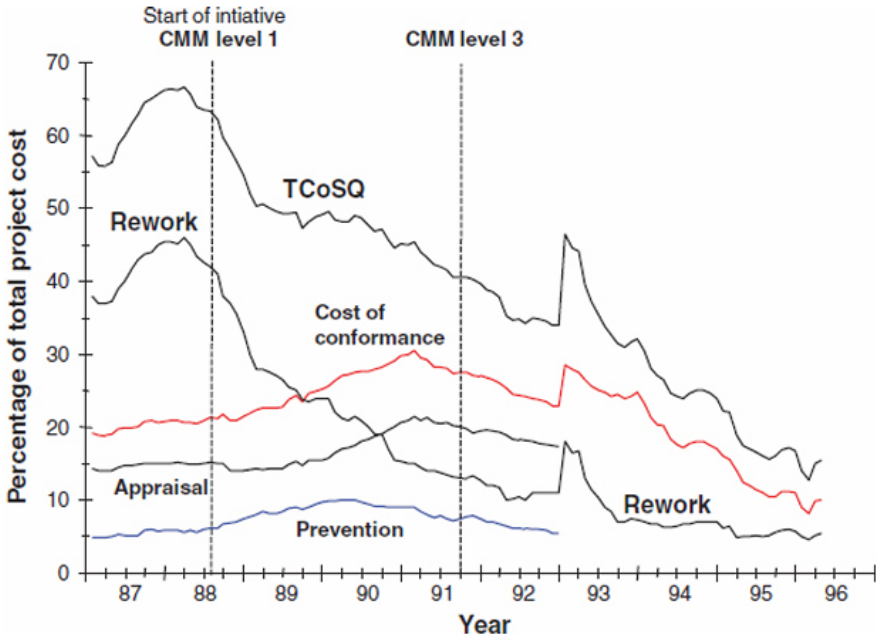


Figure 2.4 Data on software quality costs [HAL 96].

Another study on quality costs published by Krasner proposes that the rework costs vary between 15% and 25% of the development costs for a level 3 maturity organization (see Table 2.2)

The study by Dion concluded that quality costs are closely linked to the implementation of mature processes. Another study on quality costs published by Krasner indicates that rework costs range between 15% and 25% of development costs for organizations at level 3 maturity (see Table 2.2).

Process Maturity	Description	Rework (% of Total Development Effort)
Immature	Lack of structured processes	≥ 50%
Project Controlled	Basic project-level management	25% – 50%
Defined Organizational Process	Standardized organizational processes	15% – 25%
Management by Fact	Data-driven decision-making	5% – 15%
Continuous Learning and Improvement	Focus on continuous process enhancement	≤ 5%

Table 2.2 Relationship Between the Process Maturity Characteristic and Rework [KRA 98]

This section highlights the importance of understanding quality costs and improving software processes to reduce rework and enhance competitiveness. Investing in defect prevention lowers costs, reduces failure risks, and helps companies stay ahead of competitors. Though often seen as time-wasting, defect prevention is a valuable activity that ensures better quality and long-term benefits.

Quality Culture

Tylor [TYL 10] defined culture as the **"complex whole"** of knowledge, beliefs, art, morals, laws, customs, and habits acquired socially, shaping individual and organizational behavior.

Wiegers (1996) [WIE 96], in ***Creating a Software Engineering Culture*, highlights how an organization's culture interacts with its software engineers and projects, influencing their success.**

According to Wiegers, a healthy software engineering culture comprises:

1. **Developer Commitment**: Each developer personally commits to producing quality products by systematically **using effective software engineering practices**.
2. **Managerial Support**: Managers at all levels provide an environment where software **quality is a key success factor, empowering developers to achieve this goal**.
3. **Continuous Improvement**: Team members consistently strive to improve both their processes and the quality of the products they create.

Quality culture must be built from the start by the founders and evolves as employees join (Figure 2.5). It needs to be established early and consistently reinforced. Management should **create a culture that ensures high-quality software, competitive pricing, profitability, and happy employees.**

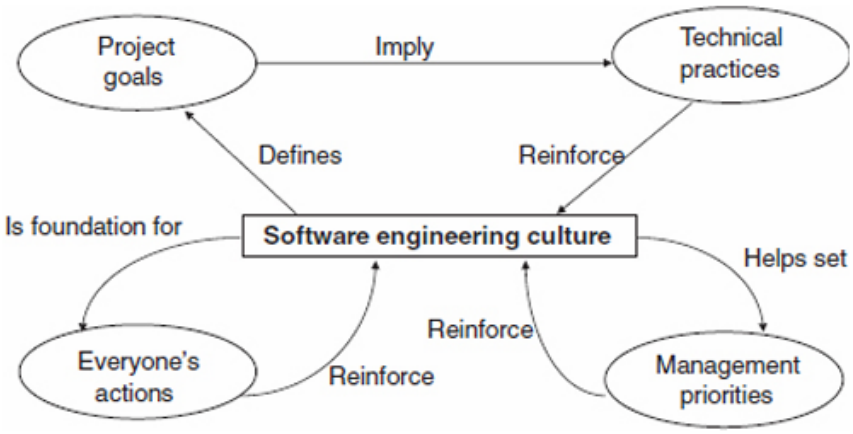


Figure 2.5 Software engineering culture.

Software engineers should care about the cultural aspects of quality because organizational change isn't achieved simply by instructing employees. Instead, the culture must evolve with

the support of change agents. A major obstacle to change is often the organization's existing culture.

Employees need to feel involved and see clear benefits from any change. If a change only satisfies a manager's whim and offers no advantage to the employee, they will be disengaged. This lack of engagement hinders cultural growth and maturity. Managing cultural change effectively is essential to achieve the desired outcomes [LAP 98].

In some companies, you may face pressure to "cut corners" on quality or skip important steps, as depicted in the comic strip (Figure 2.6). Managers or clients might push to start programming before properly identifying needs, reflecting outdated IT practices that still persist in certain organizations.

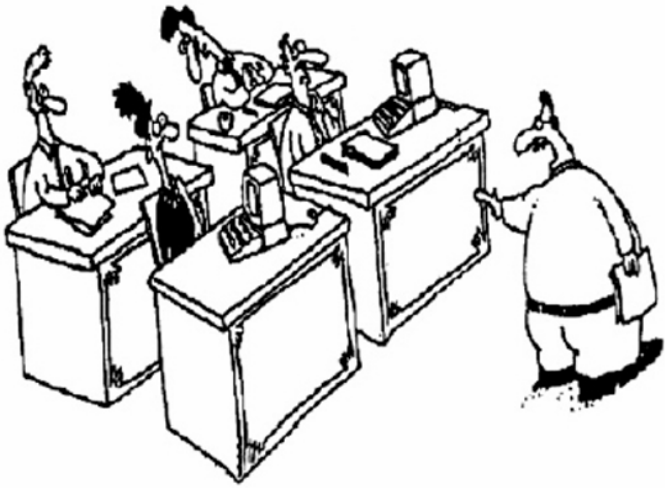


Figure 2.6 Start coding... I'll go and see what the client wants!

Figure 2.7 Dilbert is threatened and must provide an estimate on the fly. DILBERT © 2007 Scott Adams. Used By permission of UNIVERSAL UCLICK. All rights reserved.

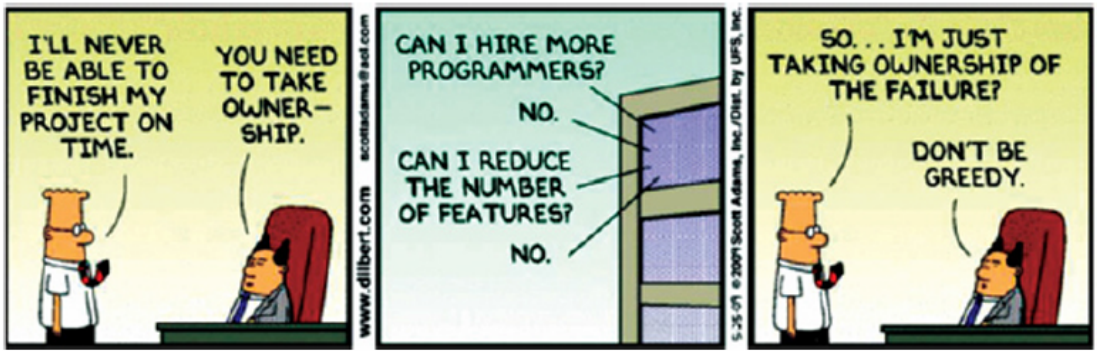


Figure 2.8 Dilbert tries to negotiate a change in his project. DILBERT © 2009 Scott Adams. Used By permission of UNIVERSAL UCLICK. All rights reserved.

Working in an environment without a strong quality culture can be challenging. When a quality culture is established, employees remain committed, even during crises. Developers and managers must adopt principles that motivate adherence to processes, balancing flexibility with maintaining quality standards. In tough situations, never compromise on quality or integrity, regardless of pressure from bosses, colleagues, or clients.

Wiegers lists four teen principles to follow to develop a culture that fosters quality
A

Here's a concise summary of the key principles for fostering software quality:

1. **Maintain Integrity**: Never compromise on quality under pressure from a boss or client.
2. **Appreciation Matters**: People need to feel their work is valued.
3. **Lifelong Learning**: Every team member is responsible for their continued education.
4. **Client Involvement**: Active client participation is critical for software quality.
5. **Shared Vision**: Align the client's vision with the development team's goals.
6. **Continuous Improvement**: Always strive to enhance development processes.
7. **Best Practices Culture**: Procedures foster a shared culture of excellence.
8. **Quality First**: Prioritizing quality leads to long-term productivity.
9. **Peer Reviews**: Peers, not clients, should identify defects.
10. **Efficient Coding**: Complete all steps before coding; code only once.
11. **Track Changes**: Manage error reports and change requests effectively.
12. **Measure and Improve**: Tracking progress helps refine processes.
13. **Pragmatic Approach**: Use reasonable methods over rigid rules.
14. **Targeted Changes**: Focus on impactful changes and start implementing them promptly.

These principles ensure high-quality outcomes and sustainable development practices.

Role of Software Quality Assurance (SQA) in the Software Development Life Cycle (SDLC)
SQA plays a critical role in ensuring that software meets defined quality standards and satisfies stakeholder needs throughout the **SDLC phases**. Below is an overview of SQA's responsibilities and activities in each phase:

---

1. Requirements Phase

- **Objective:** Ensure clarity, completeness, and correctness of requirements.
- **SQA Activities:**
  - **Review and validate requirements** for ambiguity and feasibility.
  - Ensure **traceability** of requirements to business objectives.
  - Develop checklists and standards for requirement documentation.

---

2. Design Phase

- **Objective:** Verify the accuracy and consistency of system and software design.
- **SQA Activities:**
  - Conduct design reviews and audits.
  - Validate that the design aligns with requirements.
  - Assess architecture, data flow, and algorithm effectiveness.
  - Ensure compliance with design standards.

---

3. Implementation (Coding) Phase

- **Objective:** Ensure code quality, maintainability, and adherence to standards.
- **SQA Activities:**
  - Enforce coding standards and best practices.
  - Perform static code analysis to detect potential issues early.
  - Conduct peer reviews and inspections of code.
  - Promote test-driven development (TDD) and automated unit testing.

---

4. Testing Phase

- **Objective:** Identify and fix defects in the software before release.
- **SQA Activities:**
  - Develop and review test plans, cases, and scripts.
  - Conduct various testing methods: unit, integration, system, and acceptance testing.
  - Track and analyze defect reports to ensure comprehensive fixes.
  - Validate software performance, reliability, and usability.

---

5. Deployment Phase

- **Objective:** Ensure the software is ready for release and meets deployment criteria.
- **SQA Activities:**
  - Perform final quality assurance checks (release readiness review).
  - Verify compliance with release and acceptance criteria.
  - Ensure all known defects are documented and prioritized.

---

6. Maintenance Phase

- **Objective:** Maintain software quality after deployment through updates and bug fixes.
- **SQA Activities:**
  - Monitor user feedback and defect reports.
  - Ensure quality of updates and patches through regression testing.
  - Conduct root cause analysis for recurring issues.

Key Roles of SQA Across the SDLC

1. **Prevention:** Prevent defects through proactive measures like reviews, standards enforcement, and training.
2. **Detection:** Identify defects early through thorough testing and audits.
3. **Correction:** Facilitate timely resolution of defects.
4. **Continuous Improvement:** Use metrics and feedback to refine processes and reduce defect rates in future projects.

Benefits of SQA in the SDLC

- **Cost Reduction:** Early defect detection reduces rework costs.
- **Enhanced Reliability:** Ensures software performs as expected in real-world scenarios.
- **Higher User Satisfaction:** Delivering quality software increases client and end-user confidence.
- **Regulatory Compliance:** Adherence to industry standards and legal requirements.
- **Competitiveness:** High-quality software enhances the organization's market reputation and competitiveness.

Conclusion:

SQA integrates quality into every phase of the SDLC, **ensuring that software meets high standards and aligns with stakeholder expectations**. Its role is essential in delivering reliable, efficient, and maintainable software while optimizing resources and minimizing risks.