



You:

Software reviews are a “filter” for the software process. That is, reviews are applied at various points during software engineering and serve to uncover errors and defects that can then be removed. Software reviews “purify” software engineering work products, including requirements and design models, code, and testing data. Freedman and Weinberg [Fre90] discuss the need for reviews this way: Technical work needs reviewing for the same reason that pencils need erasers: To err is human. The second reason we need technical reviews is that although people are good at catching some of their own errors, large classes of errors escape the originator more easily than they escape anyone else. The review process is, therefore, the answer to the prayer of Robert Burns: O wad some power the giftie give us to see ourselves as other see us. A review—any review—is a way of using the diversity of a group of people to:

1. Point out needed improvements in the product of a single person or team;
2. Confirm those parts of a product in which improvement is either not desired or not needed;
3. Achieve technical work of more uniform, or at least more predictable, quality than can be achieved without reviews, in order to make technical work more manageable.

Many different types of reviews can be conducted as part of software engineering. Each has its place. An informal meeting around the coffee machine is a form of review, if technical problems are discussed. A formal presentation of software architecture to an audience of customers, management, and technical staff is also a form of review. In this book, however, I focus on technical or peer reviews, exemplified by casual reviews, walkthroughs, and inspections. A technical review (TR) is the most effective filter from a quality control standpoint. Conducted by software engineers (and others) for software engineers, the TR is an effective means for uncovering errors and improving software quality.

ANSWER THE QUESTION : Discuss software review techniques.



Software reviews are crucial elements in the software engineering process, serving as a filter to uncover errors and defects at various stages of development. These reviews, applied to work products such as requirements, design models, code, and testing data, act as a means of purifying and improving the quality of the software. Here are key points and techniques associated with software reviews:

1. Purpose of Software Reviews:

- **Error Detection:** Software reviews aim to uncover errors and defects in work products, similar to how pencils need erasers because making mistakes is human.
- **Defect Amplification:** Reviews help in catching and addressing errors early in the process, preventing them from amplifying and causing more significant issues later in the project.

2. Need for Technical Reviews:

- **Human Limitations:** People are good at catching some of their own errors, but certain types of errors are more likely to escape the original creator. Reviews involve others to catch errors that might be overlooked by the originator.

3. Diversity in Review Process:

- **Group Dynamics:** A review leverages the diversity of a group of people to point out needed improvements in a product. It allows different perspectives to identify potential issues.

4. Types of Reviews:

- **Informal Reviews:** These may occur in casual settings, such as discussions around the coffee machine, where technical problems are informally addressed.
- **Formal Reviews:** Examples include walkthroughs and inspections, which are more structured and involve a systematic examination of work products.

5. Review Steps:

- **Planning:** Setting the objectives and scope of the review.
- **Preparation:** Reviewers familiarize themselves with the work product.
- **Structuring the Meeting:** Conducting the review session with a defined structure.
- **Noting Errors:** Identifying and documenting issues or errors found during the review.
- **Making Corrections:** Corrections are made outside the review, addressing the identified issues.
- **Verification:** Ensuring that corrections have been implemented correctly.

6. Participants in Reviews:

- **Software Engineers:** Reviews are typically performed by software engineers, also known as peer reviews, in collaboration with their colleagues.

7. Importance of Reviews:

- **Early Error Detection:** Finding errors early in the process is less expensive to correct.
- **Error Amplification Prevention:** Unaddressed minor errors early in the process can lead to significant issues later.
- **Time Saving:** Reviews reduce the amount of rework needed late in the project, saving time and resources.

8. Output of Reviews:

- **List of Issues/Errors:** The primary output is a list of identified issues and errors.
- **Technical Status Indication:** Reviews provide an indication of the technical status of the work product.

9. Ensuring Effectiveness:

- **Selection of Review Type:** Choosing the appropriate type of review based on the development culture.

- **Guideline Adherence:** Following guidelines that lead to successful reviews.
- **Quality Improvement:** If reviews result in higher-quality software, the process is considered successful.

In summary, software reviews, particularly technical or peer reviews, play a vital role in improving software quality by detecting and addressing errors early in the development process. The diversity of perspectives in a review helps ensure a more thorough examination of work products.



<https://www.geeksforgeeks.org/levels-of-software-testing/>



I don't have direct access to external websites, including the one you provided (<https://www.geeksforgeeks.org/levels-of-software-testing/>). However, I can certainly provide information on the general levels of software testing based on my training data up to January 2022.

Software testing is a critical phase in the software development life cycle that ensures the quality, reliability, and correctness of a software product. Testing is conducted at different levels of the development process. Here are common levels of software testing:

1. Unit Testing:

- Focuses on testing individual units or components of a software application in isolation.
- Ensures that each unit functions as intended.
- Developers often perform unit testing during the coding phase.

2. Integration Testing:

- Tests the interactions between different units or modules to ensure they work together as expected.
- Detects interface defects and ensures that integrated components cooperate correctly.

3. System Testing:

- Involves testing the complete and integrated software product.
- Validates that the entire system meets specified requirements.
- Tests functionalities, performance, security, and other aspects of the system.

4. Acceptance Testing:

- Determines whether the software meets the customer's expectations and requirements.
- Includes User Acceptance Testing (UAT), where end-users validate the system in a real-world scenario.

5. Regression Testing:

- Ensures that new code changes do not negatively impact existing functionalities.
- Performed to catch unexpected side effects of code modifications.

6. Performance Testing:

- Evaluates how well the system performs under different conditions, such as high load or stress.
- Includes load testing, stress testing, and scalability testing.

7. Security Testing:

- Identifies vulnerabilities and weaknesses in the system's security features.
- Ensures that the software is resistant to unauthorized access and data breaches.

8. Usability Testing:

- Assesses the user-friendliness of the software.
- Focuses on the overall user experience, ease of use, and user interface design.

9. Compatibility Testing:

- Verifies that the software functions correctly across different environments, devices, and browsers.
- Ensures compatibility with various operating systems and hardware configurations.

10. Alpha and Beta Testing:

- Alpha testing is performed by the internal development team before releasing the software to a few external users.
- Beta testing involves releasing the software to a larger group of external users to collect feedback.

Each level of testing serves a specific purpose in the software development process, contributing to the overall quality and reliability of the software product. Testing is an iterative process, and defects found during testing are addressed and retested to ensure a robust and error-free software release.



You:

<https://www.geeksforgeeks.org/differences-between-black-box-testing-vs-white-box-testing/>



ChatGPT:

I don't have direct access to external websites, including the one you provided (<https://www.geeksforgeeks.org/differences-between-black-box-testing-vs-white-box-testing/>). However, I can certainly provide information on the differences between black-box testing and white-box testing based on my training data up to January 2022.

Black Box Testing:

1. Definition:

- Black-box testing is a testing technique where the tester does not have any knowledge of the internal workings, logic, or code structure of the software.

2. Focus:

- It focuses on the functional requirements and specifications of the software.

3. Tester's Knowledge:

- Testers are only aware of the inputs, expected outputs, and the functionality of the software being tested.

4. Testing Levels:

- It can be applied at all levels of testing, including unit, integration, system, and acceptance testing.

5. Test Cases:

- Test cases are designed based on the software's specifications without considering the internal code structure.

6. Testing Approach:

- The tester interacts with the software as an end-user, without any knowledge of the internal implementation.

7. Applicability:

- Ideal for validating the correctness of functionalities, user interface, and overall system behavior.

White Box Testing:**1. Definition:**

- White-box testing, also known as clear-box or glass-box testing, is a testing technique where the tester has complete knowledge of the internal workings, logic, and code structure of the software.

2. Focus:

- It focuses on the internal logic, code paths, and the overall structure of the software.

3. Tester's Knowledge:

- Testers are aware of the internal code, algorithms, and data structures used in the software.

4. Testing Levels:

- It is primarily applied at the unit and integration testing levels.

5. Test Cases:

- Test cases are designed based on the internal code structure, including paths, conditions, and branches.

6. Testing Approach:

- The tester designs test cases based on the knowledge of how the software is implemented, allowing for thorough coverage of code paths.

7. Applicability:

- Ideal for validating the correctness of specific code segments, ensuring all logical paths are tested.

In summary, black-box testing is focused on validating the external behavior and functionality of the software, without knowledge of its internal structure. White-box testing, on the other hand, is concerned with testing the internal logic, code paths, and overall implementation of the software. Both testing approaches are valuable and are often used in combination to ensure comprehensive test coverage.



You:

Q44 - Assignment Q45. <https://www.geeksforgeeks.org/difference-between-unit-testing-and-system-testing/>



ChatGPT:

I don't have direct access to external websites, including the one you provided (<https://www.geeksforgeeks.org/difference-between-unit-testing-and-system-testing/>). However, I

can help answer questions or provide information based on my training data up to January 2022. If you have specific questions about the differences between unit testing and system testing, please feel free to ask, and I'll do my best to assist you!



McCall, Richards, and Walters [McC77] propose a useful categorization of factors that affect software quality. These software quality factors, shown in Figure 14.1, focus on three important aspects of a software product: its operational characteristics, its ability to undergo change, and its adaptability to new environments. Referring to the factors noted in Figure 14.1, McCall and his colleagues provide the following descriptions: Correctness. The extent to which a program satisfies its specification and fulfills the customer's mission objectives. Reliability. The extent to which a program can be expected to perform its intended function with required precision. [It should be noted that other, more complete definitions of reliability have been proposed (see Chapter 25).] Efficiency. The amount of computing resources and code required by a program to perform its function. Integrity. Extent to which access to software or data by unauthorized persons can be controlled. Usability. Effort required to learn, operate, prepare input for, and interpret output of a program. 402 PART THREE QUALITY MANAGEMENT pre75977_ch14.qxd 11/27/08 5:51 PM Page 402 Maintainability. Effort required to locate and fix an error in a program. [This is a very limited definition.] Flexibility. Effort required to modify an operational program. Testability. Effort required to test a program to ensure that it performs its intended function. Portability. Effort required to transfer the program from one hardware and/or software system environment to another. Reusability. Extent to which a program [or parts of a program] can be reused in other applications—related to the packaging and scope of the functions that the program performs. Interoperability. Effort required to couple one system to another. It is difficult, and in some cases impossible, to develop direct measures² of these quality factors. In fact, many of the metrics defined by McCall et al. can be measured only indirectly. However, assessing the quality of an application using these factors will provide you with a solid indication of software quality. ANSWER THE QUESTION : Explain McCall's quality factors.



McCall, Richards, and Walters proposed a comprehensive categorization of software quality factors, which are crucial aspects of a software product. These factors are grouped into three main categories, each addressing different dimensions of software quality: operational characteristics, ability to undergo change, and adaptability to new environments. Below are explanations of McCall's software quality factors:

1. Correctness:

- **Definition:** The extent to which a program satisfies its specification and fulfills the customer's mission objectives.
- **Importance:** This factor focuses on the accuracy and precision of the software in meeting the specified requirements.

2. Reliability:

- **Definition:** The extent to which a program can be expected to perform its intended function with the required precision.
- **Note:** The definition provided is concise, and it's acknowledged that more comprehensive definitions of reliability exist (referenced to Chapter 25).

3. Efficiency:

- **Definition:** The amount of computing resources and code required by a program to perform its function.
- **Importance:** Efficiency addresses the optimization of resources, ensuring that the software performs its tasks in a resource-efficient manner.

4. Integrity:

- **Definition:** The extent to which access to software or data by unauthorized persons can be controlled.
- **Importance:** This factor emphasizes security measures to control and protect against unauthorized access to software and data.

5. Usability:

- **Definition:** Effort required to learn, operate, prepare input for, and interpret output of a program.
- **Importance:** Usability considers the user experience, including ease of learning, operation, and interpretation of the software's output.

6. Maintainability:

- **Definition:** Effort required to locate and fix an error in a program.
- **Note:** The definition provided is acknowledged as a limited one. Maintainability focuses on the ease of error detection and correction in the software.

7. Flexibility:

- **Definition:** Effort required to modify an operational program.
- **Importance:** This factor addresses the ease with which changes and modifications can be made to the software.

8. Testability:

- **Definition:** Effort required to test a program to ensure that it performs its intended function.
- **Importance:** Testability is concerned with the ease of testing the software to verify its functionality.

9. Portability:

- **Definition:** Effort required to transfer the program from one hardware and/or software system environment to another.
- **Importance:** Portability assesses the ease with which the software can be transferred and run in different environments.

10. Reusability:

- **Definition:** Extent to which a program or parts of a program can be reused in other applications, related to the packaging and scope of functions.
- **Importance:** Reusability focuses on the potential for using the software or its components in different applications, promoting efficiency and consistency.

11. Interoperability:

- **Definition:** Effort required to couple one system to another.
- **Importance:** Interoperability addresses the ability of the software to work seamlessly with other systems.

While it may be challenging to directly measure some of these quality factors, they provide a comprehensive framework for assessing and ensuring the overall quality of a software application. Evaluating a software product based on these factors offers a solid indication of its quality and helps guide improvement efforts.



The distributed nature of client-server environments, the performance issues associated with transaction processing, the potential presence of a number of different hardware platforms, the complexities of network communication, the need to service multiple clients from a centralized (or in some cases, distributed) database, and the coordination requirements imposed on the server all combine to make testing of client-server architectures and the software that resides within them considerably more difficult than stand-alone applications. In fact, recent industry studies indicate a significant increase in testing time and cost when client-server environments are developed. In general, the testing of client-server software occurs at three different levels: (1) Individual client applications are tested in a “disconnected” mode; the operation of the server and the underlying network are not considered. (2) The client software and associated server applications are tested in concert, but network operations are not explicitly exercised. (3) The complete client-server architecture, including network operation and performance, is tested.

CHAPTER 18 TESTING CONVENTIONAL APPLICATIONS 503

Hundreds, if not thousands, of GUI testing tool resources can be evaluated on the Web. A good starting point for open-source tools is www.opensourcetesting.org/functional.php. Note: “The topic of testing is one area in which a good deal of commonality exists between traditional system and client/server systems.” Kelley Bourne

WebRef Useful client-server testing information and resources can be found at www.csst-technologies.com/pre75977_ch18.qxd 11/27/08 6:12 PM Page 503

Although many different types of tests are conducted at each of these levels of detail, the following testing approaches are commonly encountered for client-server applications:

- Application function tests. The functionality of client applications is tested using the methods discussed earlier in this chapter and in Chapters 19 and 20. In essence, the application is tested in stand-alone fashion in an attempt to uncover errors in its operation.
- Server tests. The coordination and data management functions of the server are tested. Server performance (overall response time and data throughput) is also considered.
- Database tests. The accuracy and integrity of data stored by the server is tested. Transactions posted by client applications are examined to ensure that data are properly stored, updated, and retrieved. Archiving is also tested.
- Transaction tests. A series of tests are created to ensure that each class of transactions is processed according to requirements. Tests focus on the correctness of processing and also on performance issues (e.g., transaction processing times and transaction volume).
- Network communication tests. These tests verify that communication among the nodes of the network occurs correctly and that message passing, transactions, and related network traffic occur without error. Network security tests may also be conducted as part of these tests. To accomplish these testing approaches, Musa [Mus93] recommends the development of operational profiles derived from client-server usage scenarios.⁹ An operational profile indicates how different types of users interoperate with the client-server system. That is, the profiles provide a “pattern of usage” that can be applied when tests are designed and executed. For example, for a particular type of user, what percentage of transactions will be inquiries? updates? orders? To develop the operational profile, it is necessary to derive a set of scenarios that are similar to use cases (Chapters 5 and 6). Each scenario addresses who, where, what, and why. That is, who the user is, where (in the physical client-server architecture) the system interaction occurs, what the transaction is, and why it has occurred. Scenarios can be derived using requirements elicitation techniques (Chapter 5) or through less formal discussions with end users. The result, however, should be the same. Each scenario should provide an indication of the system functions that will be required to service a particular user, the order in which those functions are required, the timing and response that is expected, and the frequency with which each function is used. These data are then combined (for all users) to create the operational profile. In general, testing effort and the number of test cases to be executed are

504 PART THREE QUALITY MANAGEMENT What types

of tests are conducted for client-server systems? ? 9 It should be noted that operational profiles can be used in testing for all types of system architectures, not just client-server architecture. pre75977_ch18.qxd 11/27/08 6:12 PM Page 504 allocated to each usage ANSWER THE QUESTION : Describe various approaches for testing client-server applications.



ChatGPT:

Testing client-server applications presents unique challenges due to the distributed nature of these environments, performance issues associated with transaction processing, and the complexities of network communication. The testing of client-server software typically occurs at three different levels of detail:

1. Individual Client Applications Testing:

- **Mode:** Disconnected mode.
- **Focus:** Tests individual client applications in isolation, without considering the operation of the server and the underlying network.
- **Objective:** Uncover errors in the operation of client applications.

2. Client-Server Applications Testing:

- **Mode:** Connected mode, but network operations are not explicitly exercised.
- **Focus:** Tests the client software and associated server applications in concert without explicitly testing network operations.
- **Objective:** Ensure that the client and server components work together seamlessly.

3. Complete Client-Server Architecture Testing:

- **Mode:** Includes network operation and performance testing.
- **Focus:** Comprehensive testing of the entire client-server architecture, including network operations and performance.
- **Objective:** Validate the complete system's functionality, performance, and ability to operate in a distributed environment.

Common testing approaches for client-server applications include:

• Application Function Tests:

- **Description:** Tests the functionality of client applications using methods discussed for stand-alone applications.
- **Objective:** Uncover errors in the operation of client applications.

• Server Tests:

- **Description:** Tests the coordination and data management functions of the server, including performance evaluation.
- **Objective:** Ensure the proper functioning and performance of the server.

• Database Tests:

- **Description:** Tests the accuracy and integrity of data stored by the server, including transaction processing and archiving.
- **Objective:** Validate data storage, update, retrieval, and archiving.

• Transaction Tests:

- **Description:** Tests various classes of transactions to ensure correctness and assess performance issues such as transaction processing times and volume.
- **Objective:** Validate the processing correctness and performance of transactions.

- **Network Communication Tests:**

- **Description:** Verify correct communication among network nodes, including message passing, transactions, and network traffic.
- **Objective:** Ensure error-free network communication and may include network security tests.

To effectively conduct these testing approaches, Musa recommends the development of operational profiles derived from client-server usage scenarios. Operational profiles provide a pattern of usage for different types of users and guide the design and execution of tests. Scenarios are derived based on use cases, addressing who, where, what, and why, providing a detailed indication of system functions, order of execution, timing, response expectations, and frequency of use.

In summary, testing client-server applications involves a combination of testing at the individual client and server levels, comprehensive testing of the complete architecture, and the development of operational profiles based on usage scenarios to guide testing efforts.



You:

Top-down integration. Top-down integration testing is an incremental approach to construction of the software architecture. Modules are integrated by moving downward through the control hierarchy, beginning with the main control module CHAPTER 17 SOFTWARE TESTING STRATEGIES 459 There are some situa- tions in which you will not have the resources to do comprehensive unit testing. Select critical or complex modules and unit test only those. Taking the “big bang” approach to integration is a lazy strategy that is doomed to failure. Integrate incremen- tally, testing as you go. When you develop a project schedule, you’ll have to consider the manner in which inte- gration will occur so that components will be available when needed. pre75977_ch17.qxd 11/27/08 6:09 PM Page 459 (main program). Modules subordinate (and ultimately subordinate) to the main con- trol module are incorporated into the structure in either a depth-first or breadth-first manner. Referring to Figure 17.5, depth-first integration integrates all components on a major control path of the program structure. Selection of a major path is somewhat arbitrary and depends on application-specific characteristics. For example, selecting the left-hand path, components M1, M2 , M5 would be integrated first. Next, M8 or (if necessary for proper functioning of M2) M6 would be integrated. Then, the central and right-hand control paths are built. Breadth-first integration incorporates all com- ponents directly subordinate at each level, moving across the structure horizontally. From the figure, components M2, M3, and M4 would be integrated first. The next con- trol level, M5, M6, and so on, follows. The integration process is performed in a series of five steps: 1. The main control module is used as a test driver and stubs are substituted for all components directly subordinate to the main control module. 2. Depending on the integration approach selected (i.e., depth or breadth first), subordinate stubs are replaced one at a time with actual components. 3. Tests are conducted as each component is integrated. 4. On completion of each set of tests, another stub is replaced with the real component. 5. Regression testing (discussed later in this section) may be conducted to ensure that new errors have not been introduced. The process continues from step 2 until the entire program structure is built. 460 PART THREE QUALITY MANAGEMENT What are the steps for top-down integration? ? M1 M2 M3 M5 M6 M7 M8 M4 FIGURE 17.5 Top-down integration pre75977_ch17.qxd 11/27/08 6:09 PM Page 460 The top-down integration strategy verifies major control or decision points early in the test process. In a “well-factored” program structure, decision making occurs at upper levels in the hierarchy and is therefore encountered first. If major control problems do exist, early recognition is essential. If depth-first integration is selected, a complete function of the software may be

implemented and demonstrated. Early demonstration of functional capability is a confidence builder for all stakeholders. Top-down strategy sounds relatively uncomplicated, but in practice, logistical problems can arise. The most common of these problems occurs when processing at low levels in the hierarchy is required to adequately test upper levels. Stubs replace low-level modules at the beginning of top-down testing; therefore, no significant data can flow upward in the program structure. As a tester, you are left with three choices: (1) delay many tests until stubs are replaced with actual modules, (2) develop stubs that perform limited functions that simulate the actual module, or (3) integrate the software from the bottom of the hierarchy upward. The first approach (delay tests until stubs are replaced by actual modules) can cause you to lose some control over correspondence between specific tests and incorporation of specific modules. This can lead to difficulty in determining the cause of errors and tends to violate the highly constrained nature of the top-down approach. The second approach is workable but can lead to significant overhead, as stubs become more and more complex. The third approach, called bottom-up integration is discussed in the paragraphs that follow.

Bottom-up integration. Bottom-up integration testing, as its name implies, begins construction and testing with atomic modules (i.e., components at the lowest levels in the program structure). Because components are integrated from the bottom up, the functionality provided by components subordinate to a given level is always available and the need for stubs is eliminated. A bottom-up integration strategy may be implemented with the following steps:

1. Low-level components are combined into clusters (sometimes called builds) that perform a specific software subfunction.
2. A driver (a control program for testing) is written to coordinate test case input and output.
3. The cluster is tested.
4. Drivers are removed and clusters are combined moving upward in the program structure.

Integration follows the pattern illustrated in Figure 17.6. Components are combined to form clusters 1, 2, and 3. Each of the clusters is tested using a driver (shown as a dashed block). Components in clusters 1 and 2 are subordinate to Ma. Drivers D1 and D2 are removed and the clusters are interfaced directly to Ma. Similarly, driver D3 for cluster 3 is removed prior to integration with module Mb. Both Ma and Mb will ultimately be integrated with component Mc, and so forth.

CHAPTER 17 SOFTWARE TESTING STRATEGIES
461 What problems may be encountered when top-down integration is chosen? ? What are the steps for bottom-up integration? ? Bottom-up integration eliminates the need for complex stubs. pre75977_ch17.qxd 11/27/08 6:09 PM Page 461 As integration moves upward, the need for separate test drivers lessens. In fact, if the top two levels of program structure are integrated top down, the number of drivers can be reduced substantially and integration of clusters is greatly simplified.

ANSWER THE QUESTION : Explain top-down and bottom-up integration testing with an example.



Top-Down Integration Testing:

Top-down integration testing is an incremental approach to constructing the software architecture. The process involves integrating modules by moving downward through the control hierarchy, starting with the main control module. There are two variations of top-down integration: depth-first and breadth-first.

1. Depth-First Integration:

- Integrates all components on a major control path of the program structure.
- Major paths are selected based on application-specific characteristics.
- For example, components M1, M2, and M5 might be integrated first, followed by other control paths.

2. Breadth-First Integration:

- Incorporates all components directly subordinate at each level, moving horizontally across the structure.
- For example, components M2, M3, and M4 might be integrated first, followed by the next control level (M5, M6, etc.).

Steps for Top-Down Integration:

1. Use the main control module as a test driver, with stubs replacing subordinate components initially.
2. Replace stubs with actual components one at a time based on the selected integration approach.
3. Conduct tests for each integrated component.
4. After completing tests for a set of components, replace another stub with the corresponding real component.
5. Perform regression testing to ensure new errors haven't been introduced.
6. Continue the process until the entire program structure is built.

Advantages of Top-Down Integration:

- Verifies major control or decision points early.
- Demonstrates functional capability early, boosting confidence.

Challenges of Top-Down Integration:

- Logistical issues may arise when processing at low levels is required to adequately test upper levels.
- Choices involve delaying tests, developing complex stubs, or integrating from the bottom up.

Bottom-Up Integration Testing:

Bottom-up integration testing starts with atomic modules (components at the lowest levels in the program structure) and gradually builds up. This approach eliminates the need for stubs as components are integrated from the bottom up.

Steps for Bottom-Up Integration:

1. Combine low-level components into clusters (or builds) that perform specific subfunctions.
2. Write a driver (control program for testing) to coordinate test case input and output for each cluster.
3. Test each cluster independently.
4. Remove drivers and combine clusters moving upward in the program structure.

Example: Consider a software system with modules M1, M2, M3, M4, M5, M6, M7, and M8. In a top-down approach, you might start with the main control module (M1) and integrate subordinate modules one by one. In a bottom-up approach, you might first combine atomic modules into clusters (e.g., {M2, M3}, {M5, M6, M7}, {M4, M8}) and then test and integrate these clusters moving upward.

Advantages of Bottom-Up Integration:

- Functionality provided by components subordinate to a given level is always available.
- Eliminates the need for complex stubs.

Challenges of Bottom-Up Integration:

- Logistical issues may arise when the number of separate test drivers is reduced.
- Integration of clusters may become more complex as you move upward in the program structure.

Comparison:

- Top-down focuses on major control points early and demonstrates functional capability early.
- Bottom-up starts with atomic modules and eliminates the need for complex stubs.

Choosing between top-down and bottom-up integration depends on project characteristics, and sometimes a hybrid approach may be used. Each approach has its advantages and challenges, and the decision is often influenced by the specific requirements and constraints of the software development project.



The basis path testing technique described in Section 18.4 is one of a number of techniques for control structure testing. Although basis path testing is simple and highly effective, it is not sufficient in itself. In this section, other variations on control structure testing are discussed. These broaden testing coverage and improve the quality of white-box testing.

18.5.1 Condition Testing

Condition testing [Tai89] is a test-case design method that exercises the logical conditions contained in a program module. A simple condition is a Boolean variable or a relational expression, possibly preceded with one NOT (\neg) operator. A relational expression takes the form $E1 <\text{relational-operator}> E2$ where $E1$ and $E2$ are arithmetic expressions and $<\text{relational-operator}>$ is one of the following: $=$, \neq , $<$, $>$, \leq , \geq , (nonequality), $!$, or \cdot . A compound condition is composed of two or more simple conditions, Boolean operators, and parentheses. We assume that Boolean operators allowed in a compound condition include OR (\vee), AND (\wedge), and NOT (\neg). A condition without relational expressions is referred to as a Boolean expression. If a condition is incorrect, then at least one component of the condition is incorrect. Therefore, types of errors in a condition include Boolean operator errors

492 PART THREE QUALITY MANAGEMENT What is a graph matrix and how do I extend it for use in testing? quote: "Paying more attention to running tests than to designing them is a classic mistake." Brian Marick Errors are much more common in the neighborhood of logical conditions than they are in the locus of sequential processing statements.

pre75977_ch18.qxd 11/27/08 6:12 PM Page 492 (incorrect/missing/extra Boolean operators), Boolean variable errors, Boolean parenthesis errors, relational operator errors, and arithmetic expression errors. The condition testing method focuses on testing each condition in the program to ensure that it does not contain errors.

18.5.2 Data Flow Testing

The data flow testing method [Fra93] selects test paths of a program according to the locations of definitions and uses of variables in the program. To illustrate the data flow testing approach, assume that each statement in a program is assigned a unique statement number and that each function does not modify its parameters or global variables. For a statement with S as its statement number, $\text{DEF}(S) \{X \mid \text{statement } S \text{ contains a definition of } X\}$ $\text{USE}(S) \{X \mid \text{statement } S \text{ contains a use of } X\}$ If statement S is an if or loop statement, its DEF set is empty and its USE set is based on the condition of statement S . The definition of variable X at statement S is said to be live at statement S' if there exists a path from statement S to statement S' that contains no other definition of X . A definition-use (DU) chain of variable X is of the form $[X, S, S']$, where S and S' are statement numbers, X is in $\text{DEF}(S)$ and $\text{USE}(S')$, and the definition of X in statement S is live at statement S' . One simple data flow testing strategy is to require that every DU chain be covered at least once. We refer to this strategy as the DU testing strategy. It has been shown that DU testing does not guarantee the coverage of all branches of a program. However, a branch is not guaranteed to be covered by DU testing only in rare situations such as if-then-else constructs in which the then part has no definition of any variable and the else part does not exist. In this situation, the else branch of the if statement is not necessarily covered by DU testing.

18.5.3 Loop Testing

Loops are the cornerstone for the vast majority of all algorithms implemented in software. And yet, we often pay them little heed while conducting software tests. Loop testing is a white-box testing technique that focuses exclusively on the validity of loop constructs. Four different classes of loops [Bei90] can be defined: simple loops, concatenated loops, nested loops, and unstructured loops (Figure 18.7). Simple loops. The following set of tests can be applied to simple loops, where n is the maximum number of allowable passes through the loop.

1. Skip the loop entirely.
2. Only one pass through the loop.
3. Two passes through the loop.

CHAPTER 18 TESTING CONVENTIONAL APPLICATIONS 493 It is unrealistic to assume that data flow testing will be used extensively when testing a large system.

However, it can be used in a targeted fashion for areas of software that are suspect. Note: "Good testers are masters at noticing 'something funny' and acting on it." Brian Marick

pre75977_ch18.qxd 11/27/08 6:12 PM Page 493

4. m passes through the loop where $m \neq n$. 5. $n \neq 1$, $n, n \neq 1$ passes through the loop. Nested loops. If we were to extend the test approach for simple loops to nested loops, the number of possible tests would grow geometrically as the level of nesting increases. This would result in an impractical number of tests. Beizer [Bei90] suggests an approach that will help to reduce the number of tests:

1. Start at the innermost loop. Set all other loops to minimum values.
2. Conduct simple loop tests for the innermost loop while holding the outer loops at their minimum iteration parameter (e.g., loop counter) values. Add other tests for out-of-range or excluded values.
3. Work outward, conducting tests for the next loop, but keeping all other outer loops at minimum values and other nested loops to "typical" values.
4. Continue until all loops have been tested.

Concatenated loops. Concatenated loops can be tested using the approach defined for simple loops, if each of the loops is independent of the other. However, if two loops are concatenated and the loop counter for loop 1 is used as the initial value for loop 2, then the loops are not independent. When the loops are not independent, the approach applied to nested loops is recommended. Unstructured loops. Whenever possible, this class of loops should be redesigned to reflect the use of the structured programming constructs (Chapter 10).

ANSWER THE QUESTION : Discuss control structure test strategies.



Control structure testing involves various techniques to design test cases that exercise different aspects of a program's control flow. Here are some control structure test strategies discussed in the provided text:

1. Basis Path Testing:

- **Description:** Basis path testing is a white-box testing technique that focuses on the logical control flow of a program. It aims to test all the linearly independent paths through the program.
- **Application:** Effective for small to medium-sized programs where it provides good coverage of control structures.

2. Condition Testing:

- **Description:** Condition testing is a test-case design method that focuses on exercising logical conditions in a program module. It considers simple conditions (Boolean variables or relational expressions) and compound conditions (combinations of simple conditions with Boolean operators).
- **Objective:** Ensures that conditions in the program do not contain errors related to Boolean operators, variables, parentheses, relational operators, or arithmetic expressions.

3. Data Flow Testing:

- **Description:** Data flow testing selects test paths based on the locations of definitions and uses of variables in the program. It involves defining sets of variables that are defined (DEF) or used (USE) at different statements.
- **Strategy:** One approach is the DU (Definition-Use) testing strategy, where each DU chain (relationship between a variable's definition and its use) is covered at least once.

4. Loop Testing:

- **Description:** Loop testing is a white-box testing technique specifically focused on the validity of loop constructs. It classifies loops into four categories: simple loops, concatenated loops, nested loops, and unstructured loops.
- **Testing Approach:**
 - For Simple Loops: Tests include skipping the loop, one pass, two passes, m passes (where m is less than or equal to the maximum allowable passes), and n-1, n, n+1 passes.
 - For Nested Loops: A systematic approach starting from the innermost loop and gradually testing the outer loops while keeping other loops at minimum or typical values.
 - For Concatenated Loops: Similar to nested loops, the approach depends on whether the loops are independent or not.
 - For Unstructured Loops: Suggests redesigning unstructured loops to use structured programming constructs.

These strategies collectively contribute to comprehensive control structure testing, ensuring that different aspects of the program's logic are thoroughly exercised and potential errors are identified. The choice of strategy may depend on the nature and complexity of the program being tested.



You:

In an interview [Ven03] published on the Web, Bertrand Meyer discusses what I call the quality dilemma: If you produce a software system that has terrible quality, you lose because no one will want to buy it. If on the other hand you spend infinite time, extremely large effort, and huge sums of money to build the absolutely perfect piece of software, then it's going to take so long to complete and it will be so expensive to produce that you'll be out of business anyway. Either you missed the market window, or you simply exhausted all your resources. So people in industry try to get to that magical middle ground where the product is good enough not to be rejected right away, such as during evaluation, but also not the object of so much perfectionism and so much work that it would take too long or cost too much to complete. It's fine to state that software engineers should strive to produce high-quality systems. It's even better to apply good practices in your attempt to do so. But the situation discussed by Meyer is real life and represents a dilemma for even the best software engineering organizations.

14.3.1 "Good Enough" Software Stated bluntly, if we are to accept the argument made by Meyer, is it acceptable to produce "good enough" software? The answer to this question must be "yes," because major software companies do it every day. They create software with known bugs and deliver it to a broad population of end users. They recognize that some of the functions and features delivered in Version 1.0 may not be of the highest quality and plan for improvements in Version 2.0. They do this knowing that some customers will complain, but they recognize that time-to-market may trump better quality as long as the delivered product is "good enough."

406 PART THREE QUALITY MANAGEMENT When you're faced with the quality dilemma (and everyone is faced with it at one time or another), try to achieve balance—enough effort to produce acceptable quality without burying the project.

pre75977_ch14.qxd 11/27/08 5:51 PM Page 406 Exactly what is "good enough"? Good enough software delivers high-quality functions and features that users desire, but at the same time it delivers other more obscure or specialized functions and features that contain known bugs. The software vendor hopes that the vast majority of end users will overlook the bugs because they are so happy with other application functionality. This idea may resonate with many readers. If you're one of them, I can only ask you to consider some of the arguments against "good enough." It is true that "good enough" may work in some application domains and for a few major software companies. After all, if a company has a large marketing budget and can convince enough people to buy version 1.0, it has succeeded in locking them in. As I noted earlier, it can argue that it will improve quality in subsequent versions. By delivering a good enough version 1.0, it has cornered the market. If you work for a small company be wary of this philosophy. When you deliver a good enough (buggy) product, you risk permanent damage to your company's reputation. You may never get a chance to deliver version 2.0 because bad buzz may cause your sales to plummet and your company to fold. If you work in

certain application domains (e.g., real-time embedded software) or build application software that is integrated with hardware (e.g., automotive software, telecommunications software), delivering software with known bugs can be negligent and open your company to expensive litigation. In some cases, it can even be criminal. No one wants good enough aircraft avionics software! So, proceed with caution if you believe that “good enough” is a short cut that can solve your software quality problems. It can work, but only for a few and only in a limited set of application domains.

14.3.2 The Cost of Quality The argument goes something like this—we know that quality is important, but it costs us time and money—too much time and money to get the level of software quality we really want. On its face, this argument seems reasonable (see Meyer’s comments earlier in this section). There is no question that quality has a cost, but lack of quality also has a cost—not only to end users who must live with buggy software, but also to the software organization that has built and must maintain it. The real question is this: which cost should we be worried about? To answer this question, you must understand both the cost of achieving quality and the cost of low-quality software. The cost of quality includes all costs incurred in the pursuit of quality or in performing quality-related activities and the downstream costs of lack of quality. To understand these costs, an organization must collect metrics to provide a baseline for the current cost of quality, identify opportunities for reducing these costs, and provide a normalized basis of comparison. The cost of quality can be divided into costs associated with prevention, appraisal, and failure.

CHAPTER 14 QUALITY CONCEPTS 407 4 A worthwhile discussion of the pros and cons of “good enough” software can be found in [Bre02]. pre75977_ch14.qxd 11/27/08 5:52 PM Page 407

Prevention costs include (1) the cost of management activities required to plan and coordinate all quality control and quality assurance activities, (2) the cost of added technical activities to develop complete requirements and design models, (3) test planning costs, and (4) the cost of all training associated with these activities. Appraisal costs include activities to gain insight into product condition the “first time through” each process. Examples of appraisal costs include:

- Cost of conducting technical reviews (Chapter 15) for software engineering work products
- Cost of data collection and metrics evaluation (Chapter 23)
- Cost of testing and debugging (Chapters 18 through 21)

Failure costs are those that would disappear if no errors appeared before or after shipping a product to customers. Failure costs may be subdivided into internal failure costs and external failure costs. Internal failure costs are incurred when you detect an error in a product prior to shipment. Internal failure costs include

- Cost required to perform rework (repair) to correct an error
- Cost that occurs when rework inadvertently generates side effects that must be mitigated
- Costs associated with the collection of quality metrics that allow an organization to assess the modes of failure

External failure costs are associated with defects found after the product has been shipped to the customer. Examples of external failure costs are complaint resolution, product return and replacement, help line support, and labor costs associated with warranty work. A poor reputation and the resulting loss of business is another external failure cost that is difficult to quantify but nonetheless very real. Bad things happen when low-quality software is produced. In an indictment of software developers who refuse to consider external failure costs, Cem Kaner [Kaner95] states: Many of the external failure costs, such as goodwill, are difficult to quantify, and many companies therefore ignore them when calculating their cost-benefit tradeoffs. Other external failure costs can be reduced (e.g. by providing cheaper, lower-quality, post-sale support, or by charging customers for support) without increasing customer satisfaction. By ignoring the costs to our customers of bad products, quality engineers encourage quality-related decision-making that victimizes our customers, rather than delighting them. As expected, the relative costs to find and repair an error or defect increase dramatically as we go from prevention to detection to internal failure to external failure costs. Figure 14.2, based on data collected by Boehm and Basili [Boehm01b] and illustrated by Cigital Inc. [Cigital07], illustrates this phenomenon. The industry average cost to correct a defect during code generation is approximately \$977 per error. The industry average cost to correct the same error if it is discovered during system testing is \$7,136 per error. Cigital Inc. [Cigital07] considers a large application that has 200 errors introduced during coding. According to industry average data, the cost of finding and correcting defects during the coding phase is \$977 per defect. Thus, the total cost for correcting the 200 “critical” defects during this phase (200 \$977) is approximately \$195,400. Industry average data shows that the cost of finding and correcting defects during the system testing phase is \$7,136 per defect. In this case,

assuming that the system testing phase revealed approximately 50 critical defects (or only 25% of those found by Cigital in the coding phase), the cost of finding and fixing those defects (50 \$7,136) would have been approximately \$356,800. This would also have resulted in 150 critical errors going undetected and uncorrected. The cost of finding and fixing these remaining 150 defects in the maintenance phase (150 \$14,102) would have been \$2,115,300. Thus, the total cost of finding and fixing the 200 defects after the coding phase would have been \$2,472,100 (\$2,115,300 + \$356,800). Even if your software organization has costs that are half of the industry average (most have no idea what their costs are!), the cost savings associated with early quality control and assurance activities (conducted during requirements analysis and design) are compelling.

14.3.3 Risks In Chapter 1 of this book, I wrote “people bet their jobs, their comforts, their safety, their entertainment, their decisions, and their very lives on computer software. It better be right.” The implication is that low-quality software increases risks for both the developer and the end user. In the preceding subsection, I discussed one of these risks (cost). But the downside of poorly designed and implemented applications does not always stop with dollars and time. An extreme example [Gag04] might serve to illustrate.

CHAPTER 14 QUALITY CONCEPTS

Requirements	Design	Coding	Testing	Maintenance
\$139	\$455	\$977	\$7,136	\$14,102

\$16,000.00 \$14,000.00 \$12,000.00 \$10,000.00 \$8,000.00 \$6,000.00 \$4,000.00 \$2,000.00 \$-

FIGURE 14.2 Relative cost of correcting errors and defects Source: Adapted from [Boe01b].

pre75977_ch14.qxd 11/27/08 5:52 PM Page 409

Throughout the month of November 2000 at a hospital in Panama, 28 patients received massive overdoses of gamma rays during treatment for a variety of cancers. In the months that followed, 5 of these patients died from radiation poisoning and 15 others developed serious complications. What caused this tragedy? A software package, developed by a U.S. company, was modified by hospital technicians to compute doses of radiation for each patient. The three Panamanian medical physicists, who “tweaked” the software to provide additional capability, were charged with second-degree murder. The U.S. company is faced with serious litigation in two countries. Gage and McCormick comment: This is not a cautionary tale for medical technicians, even though they can find themselves fighting to stay out of jail if they misunderstand or misuse technology. This also is not a tale of how human beings can be injured or worse by poorly designed or poorly explained software, although there are plenty of examples to make the point. This is a warning for any creator of computer programs: that software quality matters, that applications must be foolproof, and that—whether embedded in the engine of a car, a robotic arm in a factory or a healing device in a hospital—poorly deployed code can kill. Poor quality leads to risks, some of them very serious.

14.3.4 Negligence and Liability The story is all too common. A governmental or corporate entity hires a major software developer or consulting company to analyze requirements and then design and construct a software-based “system” to support some major activity. The system might support a major corporate function (e.g., pension management) or some governmental function (e.g., health care administration or homeland security). Work begins with the best of intentions on both sides, but by the time the system is delivered, things have gone bad. The system is late, fails to deliver desired features and functions, is error-prone, and does not meet with customer approval. Litigation ensues. In most cases, the customer claims that the developer has been negligent (in the manner in which it has applied software practices) and is therefore not entitled to payment. The developer often claims that the customer has repeatedly changed its requirements and has subverted the development partnership in other ways. In every case, the quality of the delivered system comes into question.

14.3.5 Quality and Security As the criticality of Web-based systems and applications grows, application security has become increasingly important. Stated simply, software that does not exhibit high quality is easier to hack, and as a consequence, low-quality software can indirectly increase the security risk with all of its attendant costs and problems. In an interview in ComputerWorld, author and security expert Gary McGraw comments [Wil05]: Software security relates entirely and completely to quality. You must think about security, reliability, availability, dependability—at the beginning, in the design, architecture, test, and coding phases, all through the software life cycle [process]. Even people aware

410 PART THREE QUALITY MANAGEMENT

pre75977_ch14.qxd 11/27/08 5:52 PM Page 410

of the software security problem have focused on late life-cycle stuff. The earlier you find the software problem, the better. And there are two kinds of software problems. One is bugs, which are implementation problems. The other is software flaws—architectural problems in the design. People pay too much attention to bugs and not enough on flaws. To build a secure system, you must focus on quality, and that focus must begin during design. The concepts and methods discussed in Part 2 of this book lead to a software architecture that reduces “flaws.” By eliminating

architectural flaws (thereby improving software quality), you will make it far more difficult to hack the software.

14.3.6 The Impact of Management Actions

Software quality is often influenced as much by management decisions as it is by technology decisions. Even the best software engineering practices can be subverted by poor business decisions and questionable project management actions. In Part 4 of this book I discuss project management within the context of the software process. As each project task is initiated, a project leader will make decisions that can have a significant impact on product quality.

Estimation decisions.

As I note in Chapter 26, a software team is rarely given the luxury of providing an estimate for a project before delivery dates are established and an overall budget is specified. Instead, the team conducts a “sanity check” to ensure that delivery dates and milestones are rational. In many cases there is enormous time-to-market pressure that forces a team to accept unrealistic delivery dates. As a consequence, shortcuts are taken, activities that lead to higher-quality software may be skipped, and product quality suffers. If a delivery date is irrational, it is important to hold your ground. Explain why you need more time, or alternatively, suggest a subset of functionality that can be delivered (with high quality) in the time allotted.

Scheduling decisions.

When a software project schedule is established (Chapter 27), tasks are sequenced based on dependencies. For example, because component A depends on processing that occurs within components B, C, and D, component A cannot be scheduled for testing until components B, C, and D are fully tested. A project schedule would reflect this. But if time is very short, and A must be available for further critical testing, you might decide to test A without its subordinate components (which are running slightly behind schedule), so that you can make it available for other testing that must be done before delivery. After all, the deadline looms. As a consequence, A may have defects that are hidden, only to be discovered much later. Quality suffers.

Risk-oriented decisions.

Risk management (Chapter 28) is one of the key attributes of a successful software project. You really do need to know what might go wrong and establish a contingency plan if it does. Too many software teams prefer blind optimism, establishing a development schedule under the assumption that nothing will go wrong. Worse, they don't have a way of handling things that do go wrong. As a consequence, when a risk becomes a reality, chaos reigns, and as the degree of craziness rises, the level of quality invariably falls.

CHAPTER 14

QUALITY CONCEPTS

411 pre75977_ch14.qxd 11/27/08 5:52 PM Page 411

The software quality dilemma can best be summarized by stating Meskimen's Law—There's never time to do it right, but always time to do it over again. My advice: taking the time to do it right is almost never the wrong decision.

ANSWER THE QUESTION : Explain software quality dilemma short



The software quality dilemma, as discussed by Bertrand Meyer, revolves around the challenge of finding a balance between producing a software system with perfect quality, which is time-consuming and expensive, and delivering a system quickly that may have lower quality but is "good enough" for initial use. The dilemma is particularly relevant in the software industry where time-to-market is crucial.

"Good Enough" Software:

- Acceptance of producing "good enough" software is acknowledged in the industry.
- Companies may release software with known bugs in Version 1.0, planning improvements in subsequent versions.
- The focus is on delivering functionality that users desire, even if some features contain known bugs.
- This approach allows for quicker releases, meeting market demands and avoiding exhaustive perfectionism.

Quality vs. Cost:

- The argument is that achieving perfect software quality requires significant time and resources.
- Lack of quality also incurs costs, both to end users and the software organization.
- The cost of quality includes prevention, appraisal, and failure costs, with the latter being more expensive to address.
- Early quality control and assurance activities during requirements analysis and design are cost-effective compared to addressing issues later in the development process.

Risks and Negligence:

- Poorly designed and implemented software can lead to serious risks, including financial losses, damage to reputation, and, in extreme cases, harm to individuals.
- Negligence in software development, such as delivering buggy software, can result in permanent damage to a company's reputation.
- In certain application domains, delivering software with known bugs can be negligent, opening the door to litigation.

Security and Quality:

- Software quality is directly related to security, with poorly designed and low-quality software being more susceptible to security breaches.
- Focusing on quality, particularly addressing architectural flaws during the design phase, contributes to building more secure systems.

Impact of Management Actions:

- Management decisions, such as estimation, scheduling, and risk management, significantly influence software quality.
- Unrealistic schedules, shortcuts, and poor risk management can compromise the quality of the delivered product.

In summary, the software quality dilemma highlights the challenge of achieving an optimal balance between delivering software quickly and ensuring a high level of quality. The decision to produce "good enough" software must be made cautiously, considering potential risks and long-term consequences. Ultimately, prioritizing quality, even if it requires more time, is emphasized as a wise decision.



Software quality assurance encompasses a broad range of concerns and activities that focus on the management of software quality. These can be summarized in the following manner [Hor03]: Standards. The IEEE, ISO, and other standards organizations have produced a broad array of software engineering standards and related documents. Standards may be adopted voluntarily by a software engineering organization or imposed by the customer or other stakeholders. The job of SQA is to ensure that standards that have been adopted are followed and that all work products conform to them. Reviews and audits. Technical reviews are a quality control activity performed by software engineers for software engineers (Chapter 15). Their intent is to uncover errors. Audits are a type of review performed by SQA personnel with the intent of ensuring that quality guidelines are being followed for software engineering work. For example, an audit of the review process might be conducted to ensure that reviews are being performed in a manner that will lead to the highest likelihood of uncovering errors. Testing. Software testing (Chapters 17 through 20) is a quality control function that has one primary goal—to find errors. The job of SQA is to ensure that testing is properly planned and efficiently conducted so that it has the highest likelihood of achieving its primary goal. Error/defect collection and analysis. The only way to improve is to measure how you're doing. SQA collects and analyzes error and defect data to better understand how errors are

introduced and what software engineering activities are best suited to eliminating them. Change management. Change is one of the most disruptive aspects of any software project. If it is not properly managed, change can lead to confusion, and confusion almost always leads to poor quality. SQA ensures that adequate change management practices (Chapter 22) have been instituted. WebRef An in-depth discussion of SQA, including a wide array of definitions, can be obtained at www.swqual.com/newsletter/vol2/no1/vol2no1.html. pre75977_ch16.qxd 11/27/08 6:07 PM Page 434 Education. Every software organization wants to improve its software engineering practices. A key contributor to improvement is education of software engineers, their managers, and other stakeholders. The SQA organization takes the lead in software process improvement (Chapter 30) and is a key proponent and sponsor of educational programs. Vendor management. Three categories of software are acquired from external software vendors—shrink-wrapped packages (e.g., Microsoft Office), a tailored shell [Hor03] that provides a basic skeletal structure that is custom tailored to the needs of a purchaser, and contracted software that is custom designed and constructed from specifications provided by the customer organization. The job of the SQA organization is to ensure that high-quality software results by suggesting specific quality practices that the vendor should follow (when possible), and incorporating quality mandates as part of any contract with an external vendor. Security management. With the increase in cyber crime and new government regulations regarding privacy, every software organization should institute policies that protect data at all levels, establish firewall protection for WebApps, and ensure that software has not been tampered with internally. SQA ensures that appropriate process and technology are used to achieve software security. Safety. Because software is almost always a pivotal component of human-rated systems (e.g., automotive or aircraft applications), the impact of hidden defects can be catastrophic. SQA may be responsible for assessing the impact of software failure and for initiating those steps required to reduce risk. Risk management. Although the analysis and mitigation of risk (Chapter 28) is the concern of software engineers, the SQA organization ensures that risk management activities are properly conducted and that risk-related contingency plans have been established. In addition to each of these concerns and activities, SQA works to ensure that software support activities (e.g., maintenance, help lines, documentation, and manuals) are conducted or produced with quality as a dominant concern. QUESTION THE ANSWER : Discuss various elements of software quality assurance.



Software Quality Assurance (SQA) encompasses various elements and activities aimed at ensuring the management of software quality. Here's a discussion of the key elements mentioned in the provided text:

1. Standards:

- SQA involves the enforcement and adherence to industry standards, such as those provided by IEEE, ISO, and other standards organizations.
- Ensures that adopted standards are followed and that all work products conform to these standards.
- Standards may be voluntary or imposed by customers or stakeholders.

2. Reviews and Audits:

- Technical reviews, a quality control activity, are performed by software engineers to uncover errors.
- Audits, conducted by SQA personnel, ensure that quality guidelines are followed in software engineering work.
- For example, audits may focus on reviewing the software review process to enhance error detection.

3. Testing:

- SQA oversees the planning and efficient execution of software testing activities, which primarily aim to find errors.
- Ensures that testing is carried out effectively to increase the likelihood of identifying and correcting defects.
- Collaborates with testing teams to improve testing processes and strategies.

4. Error/Defect Collection and Analysis:

- Involves collecting and analyzing data related to errors and defects to gain insights into how they are introduced.
- Aims to understand which software engineering activities are most effective in eliminating errors.
- Data-driven analysis contributes to continuous improvement in software development processes.

5. Change Management:

- SQA ensures that effective change management practices are in place to handle disruptions caused by changes in software projects.
- Proper change management minimizes confusion, preventing negative impacts on software quality.
- Techniques from Chapter 22 may be applied to manage changes efficiently.

6. Education:

- SQA plays a role in software process improvement and is a key proponent of educational programs.
- Focuses on educating software engineers, managers, and stakeholders to enhance software engineering practices.
- Promotes a culture of continuous learning to improve overall software quality.

7. Vendor Management:

- Addresses three categories of software acquisition: shrink-wrapped packages, tailored shells, and contracted software.
- SQA suggests quality practices for vendors and ensures that quality mandates are included in contracts with external vendors.
- Aims to guarantee that high-quality software results from external acquisitions.

8. Security Management:

- Given the rise in cybercrime, SQA emphasizes policies that protect data, establish firewall protection for web applications, and ensure internal software integrity.
- Ensures the adoption of appropriate processes and technologies to achieve software security.

9. Safety:

- Acknowledges the critical role of software in human-rated systems (e.g., automotive or aircraft applications).
- Assesses the impact of software failure and takes steps to reduce risk, considering the potential catastrophic consequences of hidden defects.

10. Risk Management:

- While risk analysis and mitigation are concerns of software engineers, SQA ensures that risk management activities are properly conducted.

- Ensures the establishment of risk-related contingency plans to handle unforeseen challenges.

11. **Software Support Activities:**

- Encompasses aspects such as maintenance, help lines, documentation, and manuals.
- SQA ensures that these support activities are conducted or produced with quality as a dominant concern, contributing to overall customer satisfaction.

In summary, Software Quality Assurance involves a comprehensive set of activities and concerns that collectively contribute to the management and improvement of software quality throughout the software development lifecycle. The goal is to establish and maintain high standards, prevent errors, and continuously improve processes for enhanced software quality.
