



Review Article,

Software Testing in the SDLC: A Structured Review of Testing Levels, Principles, and Lifecycle Practices

Shafqat Khalid¹, Mahmood Ashraf^{1, *}

¹Department of Computer Science, TIMES University, Multan, 60000, Pakistan

*Corresponding Author: Mahmood Ashraf. Email: mahmoodkhn24@gmail.com

Received: 03 November 2025; Revised: 22 April 2026; Accepted: 30 July 2026; Published: 15 August 2026;

AID: 005-02-000066

<https://doi.org/10.66108/mna.v5i2.99>

Abstract: Software testing is one of the most crucial activities around Software Testing and is an essential activity in most of the stages of a Software Development Life Cycle (SDLC). Traditional presentations, however, tend to focus on testing as a last-line defect identification operation, and offer little insight into the interplay between various levels of testing, principles, phases of the testing life cycle, test tools, and current best practices. This paper has the intention of providing a structured narration review on software testing in SDLC. The review presents a synthesis of foundational literature, software-testing standards and latest research on testing levels, principles of testing, Software Testing Life Cycle (STLC), test automation, continuous testing, DevOps/CI/CD and testing assisted with AI. The findings indicate that the complementary activities at different levels of testing (unit, integration, system, and acceptance) should be planned and should be accompanied by a systematic quality assurance procedure throughout the life cycle of the testing. The deliverables of the review also highlight the potential for improving speed and repeatability of feedback via automation and continuous testing as well as the ability to take quality actions across the development and operational lifecycle using shift-left and shift-right. Then there is AI-driven exam generation, prioritization and maintenance which presents opportunities, but there's a caveat – they must be of high quality, context-appropriate and validated and require human oversight to be effective. In summary, the results align with an approach that is “botched” and “site” dependent whereby the selected test methods and tools are dependent on software characteristics, quality goals, development methodology, and risks. In addition to this analysis, the review points out research opportunities relating to automated testing, continuous testing, research into AI assisted testing and empirical comparison of testing across various SDLC environments.

Keywords: Software Testing; Software Development Life Cycle; Software Testing Life Cycle; Test Automation; Continuous testing; DevOps;

1. Introduction

Organizations are turning to software systems more and more for their operations, healthcare, finance and transportation sectors, as well as commerce and communication services and public services. With the ever-increasing complexity and interconnectedness of software, maintaining reliability and quality has become a significant software engineering issue. In this context, software testing is not only about detecting

defects at the end of the Software Development Life Cycle (SDLC), but is also a way to verify Software Requirements, perform System Validation and reduce risk throughout the entire SDLC. National Institute of Standards and Technology or NIST [1] is aware of the economic effects of poor software testing too.

SDLC is a structured approach that addresses the system's requirements, design, development, testing, installation, and operation. These activities are at the same time also interrelated with testing in the whole life-cycle, especially in highly modern environments of Agile and DevOps where automated testing can become part of the continuous integration and continuous delivery (CI/CD) pipelines. So far, the necessary precondition that software solutions should be tested well exists whenever they undergo continuous changes has been pointed out in various research works that put the focus on software quality and DevOps and their relation to continuous integration [2-4].

The ISO/IEC/IEEE 29119 standards also agree with the lifecycle-based nature of testing. The general concepts of the testing process, which are used in the ISO/IEC/IEEE 29119-1:2022 and the generic processes of testing, applicable to different software development lifecycle models (SDLCM) [5, 6] are presented in ISO/IEC/IEEE 29119-2:2022. These standards establish a basis for recognizing testing as a structured activity in quality assurance, not a check-off at the end.

While the software testing literature fills a large amount of space with discussions of various testing levels, principles, processes and the various tools that exist to help in software testing, the elements are rarely integrated together. This restricts the knowledge of how they interact with the other members of SDLC and the changes in traditional testing practices due to automation, continuous testing, DevOps/CI/CD, and artificial-intelligence supported testing [3, 4, 7, 8]. This paper therefore aims to present a structured overview on software testing in the SDLC, which integrates the various levels of testing, testing principles, Software Testing Life Cycle (STLC), testing tools, software testing automation and recent testing practices, along with their applicability, benefits and limitations.

1.1. Objective and Contribution

This paper aims to present a well-informed consolidation of software testing as a component of SDLC and discussion of the contributions of software testing levels, principles, STLC activities, STLC tools and modern software testing practices to the quality of software.

In this paper, we make three key contributions. First, it structures basic test concepts in a lifecycle perspective. Second, it discusses the purposes, applications, advantages, and disadvantages of key testing levels, processes and tools. Third, it brings together traditional testing practices to modern testing trends such as automation, continuous testing, DevOps/CI/CD and AI-driven testing.

1.2. Research Questions

The review investigates the following research questions:

- **RQ1:** What is the role of Software Testing in various stages of SDLC in software quality?
- **RQ2:** How do testing levels, testing principles, and STLC activities differ in purpose, applicability, and contribution to software quality?
- **RQ3:** What is the impact of testing tools along with current practices like automation, continuous testing, DevOps/CI/CD, and AI-assisted testing on modern software testing?

1.3. Scope of the Review

It elaborates on unit, integration, system and acceptance testing, testing principles, STLC activities, representative testing and automation tools, agile, devops, CI/CD, continuous testing and AI assisted practices as software testing is a lifecycle wide quality assurance activity. It takes into account their connection to software quality aspects like functional correctness, reliability, usability, performance and security [9, 10].

The review is not an evaluation of each testing tool or AI technique. Instead, representative approaches are explored to discover the patterns, discuss trade-offs and learn the direction going forward in software testing.

In the rest of this paper, the methodology for the review is presented in Section 2, an examination of testing as part of SDLC is presented in Section 3, the examination of testing levels is in Section 4, the examination of testing principles and quality dimensions is in Section 5, the examination of the STLC is in Section 6, the modern testing practices and critical issues are in Section 8, and the conclusion, limitation, and future research directions are in Section 9.

2. Methodology

2.1. Review Approach

This study follows a structured narrative review to collate and infuse available software-testing knowledge, peer-reviewed research, and standards for software testing in the Software Development Life Cycle (SDLC). The review explores three research questions related to the role of testing in software quality, testing levels and testing activities of the Software Testing Life Cycle (STLC), and the effects of testing automation, continuous testing, DevOps and CI/CD, and AI in software testing.

Testing processes and their interaction with the various SDLC models were organized using this as an important reference, the ISO/IEC/IEEE 29119-2:2021. The latest research that aims to link traditional and modern testing practices to DevOps, continuous integration, automation, and AI-assisted testing was seen to be a recent literature which links concepts mentioned in testing to current practices [3, 4, 8, 11].

2.2. Literature Identification and Selection

The relevant literature was accessed using the combinations of various software testing, SDLC, STLC, testing levels, software quality, test automation, continuous testing, DevOps, CI/CD and AI-assisted testing software-related terms through the IEEE Xplore, ACM Digital Library, ScienceDirect, SpringerLink and Google Scholar search engines.

Peer-reviewed journal articles, regular conference publications, recognized academic books and international standards were considered first. Any original or vintage testing concepts or principles were saved and highlighted, whereas recent research was paid prominent attention on topics like automation, continuous integration, DevOps, and artificial intelligence-based testing. Relevant sources outside of software testing and/or sources having no meaningful substance with respect to the research questions were eliminated.

2.3. Thematic and Analytical Framework

Adopting the vision of the literature, the five themes were identified: (1) testing in SDLC, (2) testing level and principles, (3) STLC activities, (4) testing tools and automation, and (5) modern practices (Agile, DevOps, CI/CD, continuous testing, and AI-assisted testing).

Four criteria (purpose, applicability, strengths, and limitations) were reviewed for each of the major testing approaches. The software quality was evaluated based on the functionality, reliability, usability, performance, and security attributes which are highly relevant for testing. This allows for the comparison of testing approaches beyond just definitions and facilitates the understanding of these approaches' practical trade-offs.

2.4. Evidence Synthesis

Thematic synthesis rather than statistical synthesis of evidence. The scientific literature as well as the standards for science assessment were applied in developing defined concepts and testing processes and to exploring current developments and current challenges. With continuous integration and DevOps, there are quality benefits and implementation issues [3, 4, 12] whilst with contemporary testing research there are

several areas highlighted as being important such as automation, test generation, machine learning and more intelligent methods [8, 11].

This synthesis thus takes into account the benefits and its limitations. For instance, there's more demand for test maintenance when using automation to improve execution speed and repeatability and more infrastructure, test selection and pipeline integration are required to make continuous testing work for a fast feedback loop.

2.5. Methodological Limitations

This review is a structured narrative review and is not a systematic review or pooled statistical evidence; the results are therefore qualitative syntheses and not pooled findings. The list of literature identified is also heavily dependent on the coverage of the selected databases. Moreover, the literature that has been reviewed may not account for developments in fast-changing domains like AI-powered testing, automation, and DevOps.

3. Software Testing within the Software Development Life Cycle

In the Software Development Life Cycle (SDLC), software testing is a result of any of the following activities: Requirements, Design, Implementation, Integration, System Behavior, Deployment, and Maintenance. ISO/IEC/IEEE 29119-2: 2021 is a generic testing process that is relevant to various SDLC models [6]. Testing is more and more moving toward iterative development, Agile and DevOps with faster feedback loops that involve automation and continuous integration [3, 4].

3.1. Testing during Requirements Analysis

Requirements analysis sets in the expected behavior of the system, and is used to start testing. In early testing, requirements are checked for completeness and consistency, ambiguity, feasibility and testability. Problems discovered at this point can decrease the amount of rework after the product has been built and prevent defects from extending to design and implementation [13, 14]. Requirements verification does not, however, validate what has to be tested later on; requirements may change and some failures may only be seen once implemented and system interaction has taken place.

3.2. Testing and Verification during Software Design

Testing helps assess architecture, interface design, interdependencies, data flows and quality requirements during design. Defect, vulnerability and coding issues can be detected during design reviews and static analysis before they are executed [7]. While static techniques can't simulate all of the runtime conditions, they can cut the cost of fixing any problems when they are first designed. Thus, the design verification should be supplemented with dynamic testing.

3.3. Implementation and Unit Testing

Unit testing tests each software in isolation and yields feedback for the software implementation quickly. Automated unit testing allows test to be repeated whenever software changes. Common practices around units' tests as well as opportunities for the improvement of automated test generation have been studied [15]. Successful unit testing does not assure correctness at the system level, though, due to interactions between components, interaction with external services, and environmental concerns there can be other failures. A higher testing level should thus be employed in association with unit testing [6, 15] in order to complement and complete the tests.

3.4. Integration Testing

Integration testing is done to test interactions between software components and can uncover interface, data exchange, dependency and synchronization issues. The most popular procedures are top-down, bottom-up, and big-bang integration, which have their respective compromises. Use top-down testing to

detect high-level control anomalies early and then must use stubs to enable top-down testing, or use bottom-up testing to quickly check low-level components, but only top-down testing will be able to validate a high-level control anomaly. small systems can be easily integrated in a big bang whereas fault localization becomes more difficult. Thus, the best approach is related to architecture, dependencies, system size, and risk [6].

Integration testing is not a repeat of unit testing; Unit testing gives evidence of single components and integration adds evidence that the single components interact correctly.

3.5. System Testing

System testing is the testing of the integrated software, as per functional and non-functional requirements. It can solve functionality, performance, reliability, usability, security, compatibility and scalability. The assessment of software quality should thus take into account several factors of software quality instead of only one indicator of software quality [9, 10].

System testing requires all of the software environment to fail; due to this fact, failures may require more effort to diagnose. The cost of system level testing can be controlled through the use of automated regression testing and the right selection of testing [16].

3.6. Acceptance Testing

Acceptance testing verifies that software meets the needs of the customers, the users and the stakeholders while ensuring that there is evidence that refers to the business and to the use of the software. Acceptance testing is in addition to developer- and tester-oriented verification [6, 17] and hence tests the usefulness in the intended context if it is technically correct.

Acceptance criteria should stem from documented requirements, business objectives and operational needs and can be assessed by the customers, the representative users or authorized stake holders.

3.7. Testing during Deployment and Continuous Integration

Testing is increasingly becoming part of continuous integration and continuous delivery/deployment in modern development. Automated tests can give instant feedback if the new integrated changes cause new regressions [4, 12].

Static can be also integrated into the CI pipelines. Zampetti et al. did a study of static-analysis usage in 20 open-source software Java projects, proving its utility for automated quality assurance and the issues that can occur due to inadequate configuration [7]. Therefore, a continuous test cycle does not mean that every test needs to run following every change; others factors need to be taken into account: test selection, test prioritization, test time, test infrastructure and test maintenance [4, 7].

3.8. Continuous Testing in DevOps

DevOps promotes teamwork, automation, continuous delivery and fast feedback. To achieve these objectives, testing plays a vital role by conducting automated quality checks across development and delivery workflows [3]. These can be either unit, integration, API, security, performance, static-analysis and regression testing, depending on the project needs.

However, satisfying continuous testing can be costly on execution time for pipelines, infrastructure, and effort in test maintenance. However unstable tests may discontinue delivery pipelines as well as undermine the confidence level of automated results. Appropriate test design, prioritization, maintenance and monitoring are therefore all necessary for effective continuous testing [4, 7].

3.9. Testing during Maintenance and Regression Testing

Testing continues because software is modified to fix bug and add functionality and adapts to new requirements and security or environment changes. Regression testing is used to verify whether or not changes have been inadvertently made to the existing functionality.

It can be expensive to maintain large test suites as they evolve, but automated regression suites can decrease the effort required for repetitive execution of tests [11]. In other words, regression-test selection is subject to a coverage/execution cost compromise. Selective- or prioritized testing can minimize compute time but these testing methods could miss tests that could expose specific regressions [16].

3.10. Shift-Left and Shift-Right Testing

It helps to introduce quality practices closer to development process by using shift-left testing practices consisting of requirement review, static analysis, unit testing and early security verification. In principle, it can shorten the lead time prior to defect discovery but can never detect failures depending on system integration, realistic workloads, production configurations and user actions.

Shift-right testing includes activities that monitor, validate in operation, test resilience and gather real-world feedback allowing to move quality initiatives to deployment and operation. Relying too heavily on production testing, however, may lead users, with both defects and operational risks. Thus shift-left needs to be paired with shift-right to complement it rather than erase it [3].

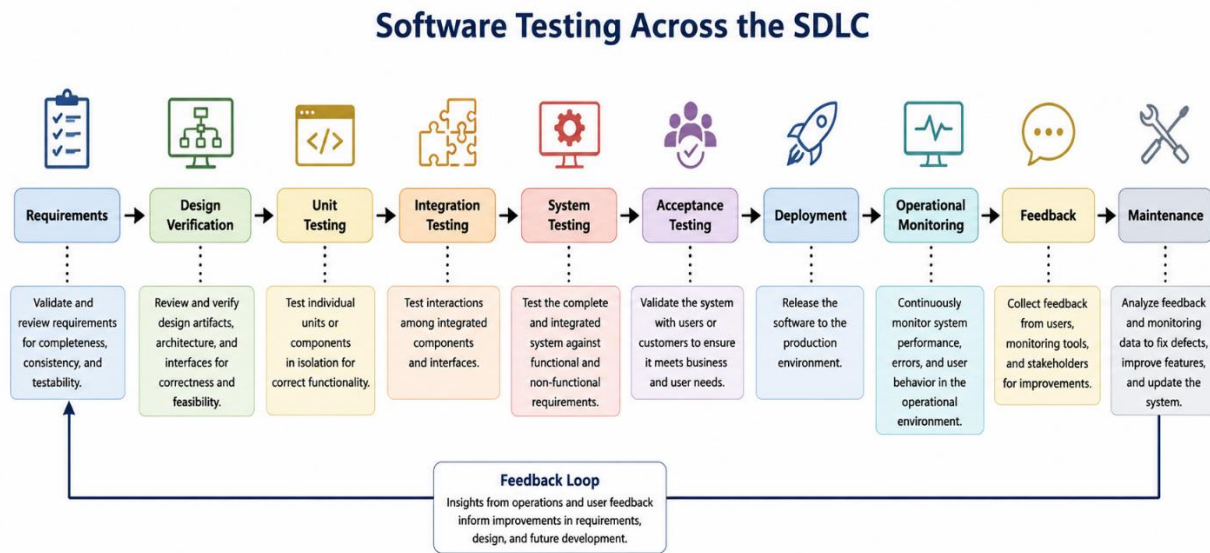


Figure 1: Software Testing Across the SDLC

Through the deployment, monitoring and maintenance processes, software testing is integrated throughout the SDLC as shown in Figure 1, with constant feedback loop between requirements and development.

The software life cycle elucidates that the software quality is not a one-time affair of testing but an iterative process of checking and revisiting.

3.11. Relationship between SDLC and STLC

The SDLC covers the entire Software development lifecycle wherein Software is planned, developed, delivered, and maintained, whereas the Software Testing Life Cycle (STLC) covers all activities pertaining to the testing of software systems for its evaluation of quality.

The two kinds of lifecycles are closely related, but serve different purposes. The activities in SDLC help to define what and how is being developed whereas the activities in STLC help to determine the way software is evaluated. Requirements feed into Test analysis and development generates builds for testing and deployment and maintenance generates contexts for acceptance, regression and operational testing. This lifecycle-oriented perspective and its corresponding testing processes are addressed by standards from the International Standards Organization (ISO), IEC, and IEEE, including ISO/IEC/IEEE 29119-2:2021.

Table 1: Relationship between SDLC Activities and Testing Activities

SDLC Activity	Corresponding Testing Activity	Primary Purpose	Typical Output
Requirements Analysis	Requirements review and test analysis	Identify ambiguity, incompleteness, and non-testable requirements	Test conditions and acceptance criteria
Design	Design review and static analysis	Identify architectural, interface, and quality risks	Review findings and refined test conditions
Implementation	Unit testing and code-level verification	Detect component-level defects	Unit-test results and defect reports
Integration	Integration testing	Verify interactions among components	Integration-test results
System Validation	System and non-functional testing	Evaluate complete system behavior	System-test results
Acceptance	Acceptance testing	Determine user and business suitability	Acceptance decision
Deployment	CI/CD validation and release testing	Verify release readiness	Release-quality evidence
Maintenance	Regression, security, performance, and operational testing	Detect unintended effects of changes	Maintenance and regression results

From Table 1, it can be seen that there are multiple SDLC activities that can be carried out for testing and the nature of testing changes based on information, risks, and software constructs present at every stage of SDLC.

3.12. Critical Evaluation of Testing across the SDLC

There are a number of benefits from integrating testing within the SDLC. Early verification can ensure needs and design issues are not more expensive to address. Rapid feedback from testing at the component level, Integration test serves to identify failure of interaction between two or more components, System test gives complete system behavior feedback, And Acceptance test serves as connection between system technical test and requirements of user and business. In fast-paced environments, continuous integration and automation in testing can provide faster feedback [3, 4, 7].

No single testing activity, however, can ensure that software is of quality. Because of this, there are limitations on requirements verification during run-time, limited visibility in unit testing, increased complexity in integration testing as the number of dependencies increases, time and infrastructure needed for system testing and relatively late detection of problems in the final form of acceptance testing. Likewise, automation increases repeatability and execution speed yet has maintenance and infrastructure costs [7, 11, 15].

The evidence then portrays a multi-layered testing approach with specific levels of information and risk assessment: Unit and integration testing, system and acceptance testing, automation and continuous testing and operational feedback. The right balance will depend on the complexity of the system and the frequency of system development, exposure to risk, regulatory requirements, quality goals, and available resources [3, 4, 6].

4. Test Levels and their Applications

Software testing is carried out across several stages in order to evaluate different parts, components and eventually the entire software and environment in which it operates. There are unit, integration, system and acceptance testing stages of the principal levels, each generating complementary evidence of software quality [5, 11].

4.1. Unit Testing

Unit testing is used to check out individual functions, methods, or classes or modules against their expected behavior [6, 7]. It is very narrow in scope and requires very short execution, so is suitable for discovering defects in early stages and frequent automated regressive test.

Common defects occur at the unit level such as logic errors, wrong calculations, boundary conditions and invalid input handling. But unit-level test is not very effective at detecting component failures, such as database, network or external service failures, or combining them together. However, high code coverage alone is not necessarily sufficient to cover all the behaviors or risks [7].

4.2. Integration Testing

Integration testing detects the interactions tests between software components or subsystems, and can also expose failures in integration, data exchange, dependency, communication failures and synchronization problems, which cannot be detected by the unit testing conducted individually. [6]

4.2.1. Integration Strategies

There are two primary strategies for integration: top down, bottom up and big bang. There is top-down integration which identifies high level control issues, but may need to use stubs. The bottom-up integration approach verifies the low-level components first, which may cause delay in high-level verification. As the systems become more complex, locating the faults becomes more difficult, and big-bang integration might be suitable for smaller systems.

Hence, the choice among integration strategies must be based on the architecture of the system, the dependency requirements, development priorities, and risk, not that one integration strategy is better than the other.

4.3. System Testing

System testing is the integration of all systems with functional and non-functional requirements. It has the ability to provide an assessment of functionality, performance, security, usability, reliability, compatibility, scalability, and resilience.

The main drawback of system testing is that it requires more infrastructure setup and more effort to conduct the testing, and if there are any failures in the system, it might be challenging to identify. It should therefore be complementary and not a substitute for lower-level testing. Automated system tests may require higher maintenance activity due to environment, interface, test data and dependency requirements, but automation can help to reduce the effort required to execute tests repeatedly. Test flakiness may also affect the confidence of automated test results [18].

4.4. Acceptance Testing

Acceptance Testing is to check and test the software to see if it meets the requirement and the expectation of the people (users, customers or other stakeholders) it intends to serve. Complements the technical testing by looking at the appropriateness of the delivered system for its operational and business environment.

Acceptance testing can reveal workflows that do not meet the criteria, lack business functionality, usability issues and/or differences between requirements and user expectations. It is not as suitable for testing for low level implementation defects, something that it is more likely possible to pick up earlier in the testing process.

4.5. Transition between Testing Levels

Testing is not wholly independent stages, but rather complementary. This progress is shown in Figure 2; the first phase is the component level verification test, followed by the verification of the correct functioning of all components before the system test or validation of the software against the users' and business requirements.

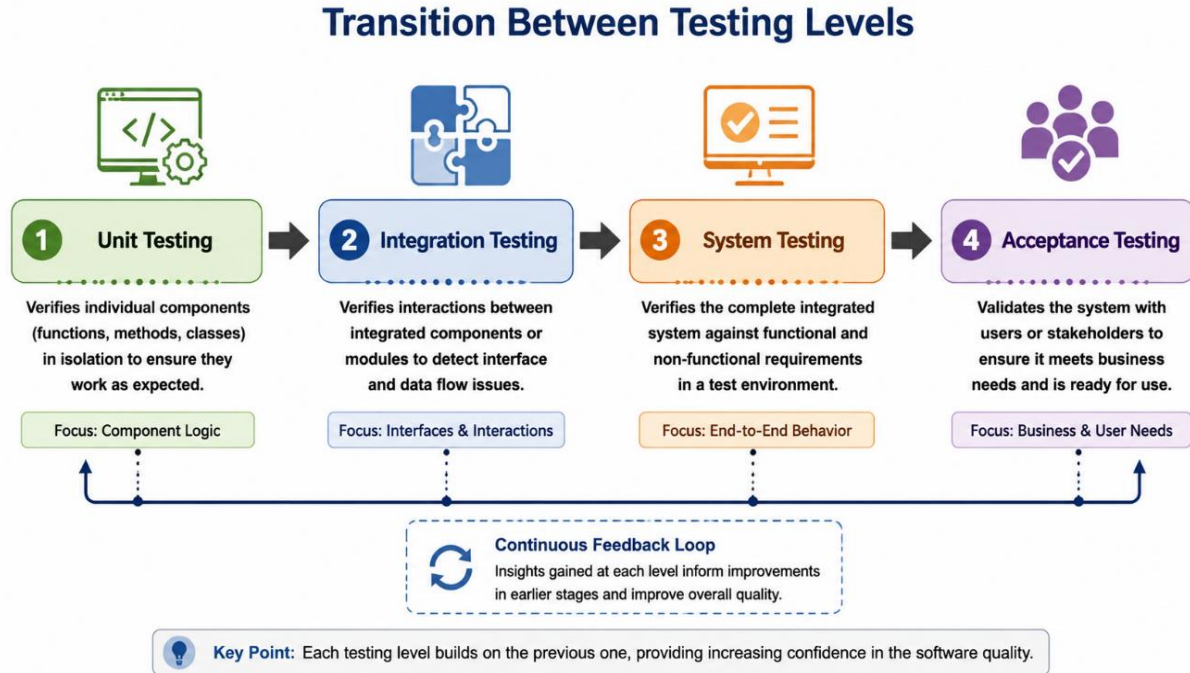


Figure 2: Transition between Testing Levels

The individual components are assessed in unit testing, how they interact in integration testing, the integrated product in system testing and the user or business requirements in the acceptance testing [5].

These levels can also be run together in modern iterative environments. A Unit and a Integration test could be done after a code change and a System and a Acceptance oriented test could be done additional checkpoints in the pipeline. There is therefore a need for test selection and prioritization, which balance quick feedback with enough coverage [4].

4.6. Defect Detection across Testing Levels

Different testing levels work well for different categories of defects. The purpose of unit testing is to design and apply tests for local implementation errors, integration testing is used for interaction and interface issues, system testing for end to end and system level faults, acceptance testing for business and user requirements.

These boundaries vary from strict, as one defect can be discovered at several levels. Even so, testing at the earliest realistic practicable level is likely to minimize the testing of failure diagnosis and minimize the effort spent on eventual failure corrections as it depends on the interaction of the system, environmental and user factors.

4.7. Comparative Analysis of Testing Levels

Table 2 provides a summary of the principal characteristics and trade-offs that are involved in these four major testing levels.

Table 2: Comparative Analysis of Software Testing Levels

Testing Level	Scope	Typical Performer	Key Techniques	Typical Defects Detected	Main Strength	Main Limitation
Unit Testing	Individual functions, methods, classes, or modules	Developers	Automated unit tests, mocks, stubs, assertions	Logic errors, boundary errors, incorrect calculations, local implementation defects	Fast feedback and relatively low execution cost	Limited visibility of component interactions
Integration Testing	Interactions between components or subsystems	Developers/Testers	Top-down, bottom-up, incremental, interface and API testing	Interface mismatches, data-exchange errors, dependency and communication failures	Detects interaction-related defects	Increasing complexity as dependencies grow
System Testing	Complete integrated system	Testers/QA teams	Functional, regression, performance, security, usability, compatibility testing	End-to-end failures and system-level functional/non-functional defects	Broad evaluation of system behavior	Higher execution and environment requirements
Acceptance Testing	System suitability for intended users/business	Customers, users, business stakeholders, QA	User acceptance, operational acceptance, scenario-based validation	Business requirement gaps, workflow problems, usability and acceptance failures	Connects technical testing with user/business needs	Often later in the lifecycle and less suitable for low-level diagnosis

The example shows that none of any testing level can give absolute security alone. Adv. for Unit testing rapid and inexpensive feedback, but narrow scope. Integration testing then reviews interactions between the components, with dependency related complexity. System testing can give a larger body of technical evaluation and would allow for more resources to be committed, although it can come after substantial work with the development effort has been made; acceptance testing would give an evaluation and validation from the user's point of view, however it can be done at the end of an extensive development effort and may not be the most thorough.

A good strategy would then be to use the testing power at the levels appropriate to their risk and expected value. Some function might need a heavier test regimen if it is high risk, and a more lightweight test regimen

if it is low risk. The distribution of the appropriate case is then context specific and not stemming from a pre-established test ratio.

4.8. Testing Levels in Modern Software Development

In modern development, testing levels and continuous integration and delivery are combined. Automated unit and integration testing can deliver instant results and for broader system and acceptance testing, these tests can be integrated into release pipelines [4]. Automation, however, does not guarantee that it will adequately cover everyone and adds in maintenance needs.

If flaky tests run differently than they did in the past, without any changes to the corresponding software, the confidence in automated pipelines is lowered, making effort in investigation higher [18]. Hence, test reliability, test time, maintainability, test environment stability and test diagnostic are all important for contemporary testing approaches to test coverage.

4.9. Critical Perspective

The four testing levels are complementary pieces of evidence but more tests do not necessarily lead to better software quality by any means. A combination of redundant or poorly designed tests and over-reliance on lower-level automation can contribute to expensive running and maintenance of tests, and leave residual user and system behavior gaps. On the other hand, the more you rely on end-to-end testing, the longer it takes and the longer will it take to figure out what is wrong.

The strategies used to achieve and balance test granularity, coverage, execution cost, maintainability and risk are therefore required for effective testing. Unfortunately, testing levels were chosen to be maximized at each level, instead of defining levels based on project context and quality objectives [4, 5].

5. Software Testing Principles and Dimensions of Quality

Application of software testing principles helps to determine how much should be spent on testing, and what the results of testing are indicating. Effective testing requires prioritization, adaptation, and the software context and risk have to be considered, as exhaustive testing of complex systems is impractical [6, 19]. The seven principles listed below should be taken as practical guidelines rather than definitions.

5.1. Testing Reveals the Presence of Defects

Testing can reveal defects; but not prove software defect free [6, 19]. A high number of tests passed does not necessarily mean that all the test items are correct. The effectiveness of a test is highly dependent on the design of the test, the selection of its input, its coverage, and its ability to expose meaningful failures. The emphasis on test-suite quality should, therefore, be established based on factors such as fault detection, coverage, reliability, maintainability, and other aspects in addition to the quantity of the test suite [11].

5.2. Exhaustive Testing is Impossible

Most non-trivial systems have too many possible inputs, execution paths, states and environments to test comprehensively [6, 19]. Requirements, risk, usage, complexity of system, and failure consequences are thus key elements for test. While automation can result in more executions of the basic test selection and prioritization cannot be replaced.

5.3. Early Testing Saves Time and Effort

Early testing involves performing requirements reviews, design verification testing, static testing and unit testing; the purpose of these types of testing is to uncover issues prior to the spread of these issues to any subsequent testing activities. Continuous integration also facilitates early feedback as quality checks are automatically performed as changes are made [4, 12]. Early testing is not a substitute for integration, system and acceptance testing, however, because some defects may not manifest themselves in the wider system context and/or meaningful use conditions.

5.4. Defect Clustering

Defect history is valuable as a tool to prioritize testing efforts because there are often relatively few components from which defects will be concentrated. But the fact that no defect was found in a component does not make a component free from defect. The historical defect information ought to be taken into account in conjunction with scoring criteria like complexity, change frequency, business criticality, security exposure, and architectural dependencies.

5.5. The Pesticide Paradox

Re-execution of a test suite, without changing anything in the software, might not be as effective in exploring previously unobserved defects, [6, 19]. Test suites should therefore grow reflecting new added functionality, changing requirements, found defects and operational conditions. While automated regression tests are important, they must be continually reviewed and maintained to maintain their usefulness [11].

5.6. Testing is Context Dependent

There is no universal testing approach for software systems. The priority of the test to be given depends on the application purpose, application architecture, application users, development model, risk and so on application security and potency requirements of regulators, quality goals and so on [5, 6]. Moreover, ISO/IEC 25010:2023 definitely reinforces this viewpoint by introducing a multi-dimensional model of software product quality that can be used for software quality evaluation in various contexts [9].

5.7. The Absence-of-Errors Fallacy

Unfortunately, even if known defects are eliminated, software is not necessarily useful or meets stakeholder requirements [6, 19]. The testing should thus fulfill not only the requirements verification but also the business needs and user needs intended. The acceptance and usability and operational testing complement technical testing by evaluating aspects that cannot be captured through conventional functional tests [9].

5.8. Software Quality Dimensions Relevant to Testing

There can be several dimensions of software Quality and software testing should focus on the properties most relevant to the software system being tested. We have a formal product-quality model as per ISO/IEC 25010, 2023, including nine product quality characteristics [9]. Five important dimensions for this particular review are functional correctness, reliability, usability, performance and security.

- **Functional correctness** is assessing the software for producing the desired functions and results. But, just being correct does not determine the overall quality as software can be correct and yet its operation may be slow, cumbersome to use, unreliable and insecure.
- **Reliability** is about the ability of a sensor to operate reliably under defined conditions. Failure, recovery, regression and operational testing are among the various forms of reliability testing, but reliability also requires consideration of workload, infrastructure and environment.
- **Usability** is a measure of the ability of intended users to use software products effectively and satisfactorily. Hence, user centered testing plays an important role, since it is not easy to use and no efficient software can be technically correct.
- **Performance** measures behavior under given workloads, such as response time, resource utilization, load, stress and scalability. Workload and test-environment properties have a high impact on the result of the test.
- **Security** means that protection against unauthorized access and misuse and other threats are assessed. Security should not be violated as the last step of SDLC but should be considered at all stages.

5.9. Relationship between Testing Principles and Quality Evaluation

The principles of testing will influence the way of testing, and the quality dimensions will influence what aspects of software quality are subject to testing. The amounts of time spent testing are influenced by context, for instance, by risk-based prioritization, and the emphasis placed on testing particular quality characteristics are influenced by the associated context as well. Both the early testing and pesticide paradox involve incremental improvement and verification, but in different ways.

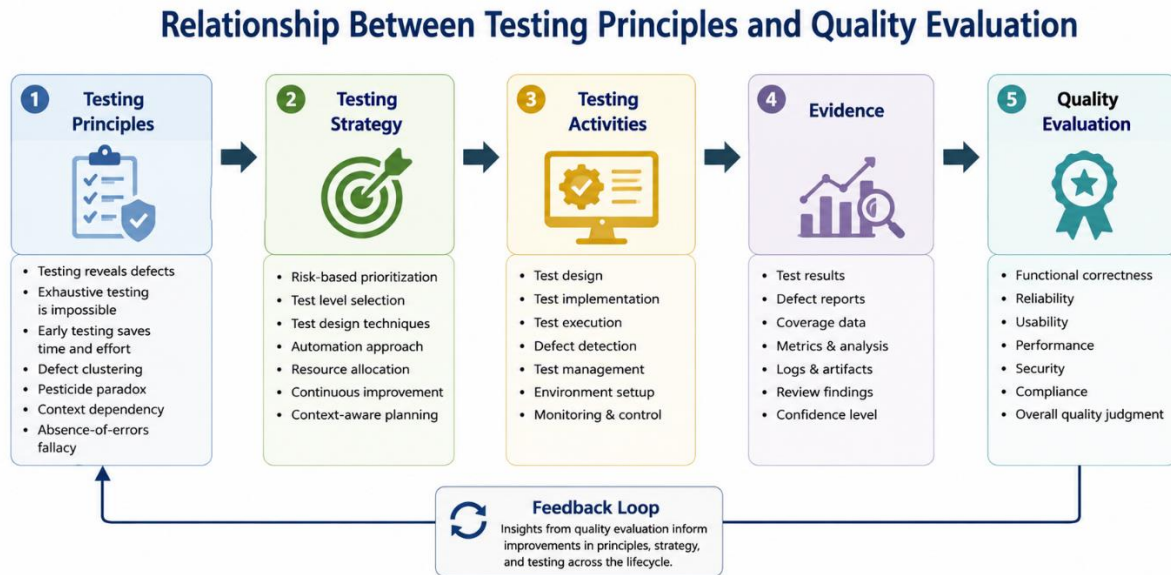


Figure 3: Relationship between Testing Principles and Quality Evaluation

It is therefore, important to determine effectiveness through a test that provides not only quantity but quality and relevance of evidence generated. Fault detection, maintainability, reliability, usability, coverage and others are also mentioned as important attributes of the quality of test suites in research with software-testing practitioners [11].

5.10. Critical Discussion

The 7 principles are obviously relevant but are not strict rules. Early testing can't detect all of the system-level problems, automation can augment repeatability but it's not necessarily compatible with a reduction in maintenance; defect clustering can have a positive benefit when it comes to prioritization but not when it comes to overlooking certain functionality.

In general, to make testing effective, it should have an evidence-informed and risk-informed prioritization, not the goal of maximum testing. Testing effort should be proportional to system complexity, system usage, quality requirements, based on history, and be related to potential negative impacts if the system fails during testing. Testing capability can be boosted with automation and continuous testing, but there are many things that require human judgment and control in test design, interpretation, maintenance, and prioritization [4, 11].

6. Software Testing Life Cycle

The Software Testing Life Cycle (STLC) is a collection of activities that are followed in planning, designing, performing and finishing software testing. Though how it is applied in an organization can differ, activities typically performed within the scope of test include requirements analysis, test planning, test case development, test environment preparation, test execution and test closing. ISO/IEC/IEEE 29119-2:2021 presents generic test processes which may be modified for various software development lifecycle models [5] to suit software projects.

- **Requirements Analysis:** Testers look at functional and non-functional requirements to understand what is to be tested and see if there are any ambiguities or missing information. The earlier the analysis is performed, the easier it is to test and the more traceability there would be from requirements to test conditions [5, 6].
- **Test Planning:** The steps used for determining the scope, objectives, resources, schedule, responsibilities, tools, risks and entry/exit criteria of the test. Planning needs are dependent on the complexity and risk of a project [5].
- **Test-Case Development:** Test Cases, Test Data, Expected Result, Automation Script are developed from identified Test Conditions & Requirements. The right technique of test design should be chosen based on the system and test purpose [10].
- **Test-Environment Setup:** The hardware, software, network configuration, databases, other tools and supporting services are set up. A representative enough environment, similar to the context of the intended test, should be used.
- **Test Execution:** The test cases are run and the results of the test are checked against the expected results. Any failures are tracked as defects and corrected software is re-tested and re-tested for regression. Confidence in automated test results is limited if there are unstable tests in automated environments that are integrated in CI/CD flow.
- **Test Closure:** Testing is performed and ended by analyzing test results, documenting problems identified but not resolved, recording lessons learnt from testing and producing a final testing report. There are some guidelines that are given for test documentation in ISO/IEC/IEEE 29119-3:2021 [20].

Table 3: Summary of STLC Phases

STLC Phase	Main Activities	Key Output
Requirements Analysis	Review requirements and identify test conditions	Test conditions
Test Planning	Define scope, resources, schedule, risks, and strategy	Test plan
Test-Case Development	Design test cases and prepare test data	Test cases/scripts
Environment Setup	Configure testing infrastructure	Test environment
Test Execution	Execute, record, retest, and report defects	Test results/defect reports
Test Closure	Evaluate results and document lessons learned	Closure report

Consider the STLC not to be rigid steps. These activities are repeatable tasks that can occur within iterations and interwoven into a continuous testing and CI/CD cycle in Agile and DevOps environments. It has been mostly useful for offering a systematic and traceable testing approach, but still flexible enough to accommodate the context of development [4, 5].

7. Software Testing Tools and Automation

A software testing tool can assist you in planning, executing, analyzing and manage the software testing activities. In modern-day software development, automation is especially noteworthy because tests should be run again and again regularly with frequent releases and as part of continuous integration processes, and they need to be run efficiently. The adoption of the tools doesn't guarantee an improvement in software quality; the choice of tools should take into account the testing objective, technology environment, software maintainability, software integration requirements, cost and the software team skills [21, 22].

The most common testing tools can be classified as per their prime purpose. Unit testing is supported by various frameworks such as JUnit, which allows developers to automate their unit components testing. Selenium is used extensively to conduct automated end-to-end testing of a Web application and interacts with a number of different browsers [23]. The Postman is frequently used for API Testing, whereas the Apache JMeter can be used for performance or load testing. Appium is an automated testing platform for mobile apps. Potential security threats and code-quality problems can be detected by Wednesday software tool SonarQube and other static-analysis tools without running the software. Machine-driven tests and analyses can be embedded in software delivery pipelines by CICD platforms like Jenkins and GitHub Actions.

The survey of the Selenium ecosystem performed by practitioners from 24 countries showed that Selenium is a pivotal framework for web-application testing, while JUnit and TestNG are widely used as unit-testing frameworks along with Selenium testing [23]. Other, more recent, empirical studies of automated software engineering tools also found that Postman, Selenium, Apache JMeter, and Junit were often used tools, including for API, web and performance testing and unit testing. The finding here suggests that the choice of tools is strongly dependent on their type when implementing testing instead of having a single best tool.

Another alternative type of automatic quality assurance is available with static-analysis tools. Zampetti et al. analyzed 20 open-source Java projects, with continuous-integration tools, and concluded that tools were employed for coding standards and to locate potential defect and vulnerability [12]. Their study also confirmed the viewpoint that incorrect configuration can lead to needless build failures, and they had to introduce some tools to the build pipelines, but without configuration and governance it would be a very futile endeavor.

Table 4: Representative Software Testing Tools

Testing Purpose	Representative Tools	Main Application	Key Consideration
Unit Testing	JUnit, TestNG	Component-level testing	Fast execution and frequent regression testing
Web/UI Testing	Selenium	Automated web testing	Browser compatibility and test maintenance
API Testing	Postman, REST-Assured	REST/API validation	Service dependencies and test-data management
Performance Testing	Apache JMeter	Load and performance testing	Realistic workload and environment configuration
Mobile Testing	Appium	Mobile application automation	Device and platform diversity
Static Analysis	SonarQube and similar tools	Code-quality and static checks	Configuration and false-positive management
CI/CD Integration	Jenkins, GitHub Actions	Automated build and test pipelines	Pipeline reliability and execution time

The benefit of automation is that you are able to run repeatable tests efficiently, and deliver fast feedback. However, the introduction of automation also adds maintenance costs to the plan – the test scripts, test data, environments, and dependencies need to change as the software changes. Several practices were identified as key for test automation success, including defining a test-automation strategy, having proper test environment, and writing good test scripts [22].

So, the testing tool should not be used solely on the basis of popularity and should be chosen based on other qualities determined by each of these factors: purpose, maintainability, capability of integrating, implementation cost and project context. Manual testing and automated testing will also require more manual than you might imagine, depending on the types of tests and tasks involved, for exploratory testing, usability testing, very variable testing, etc. may need human judgment. In software testing, there are various methodologies including manual testing and automated testing, and the ISO/IEC/IEEE 29119-3:2021 takes both types of testing into account.

8. Modern Software Testing Practices and Critical Discussion

The development of software has evolved into iterative development, repeated release, continuous integration, and automatic delivery. Such changes have made software testing more of a continuous quality-assurance activity, rather than just one performed pre-release. Modern testing practices focus on automation, quick feedback, risk prioritization, and integration with development and deployment practices [4, 24].

8.1. Agile, DevOps, and Continuous Testing

With agile development their development cycles are short and iterations delivered frequently; therefore, the testing activities happen throughout the process and not at the end. DevOps takes this to a whole new level by aligning development and operations through automation, continuous integration, continuous delivery and continuous feedback. Studies on DevOps and software quality show that testing/quality assurance plays important role in a successful DevOps adoption [24].

Continuous testing is about the automation of testing aspects into development and delivery pipelines respectively. Regardless of changes in software, unit, integration, regression, API, security and selected performance tests can be automatically run. This will help get instant feedback and can minimize the time between defect introduction and detection [4] Larger automated test suites can, however, lead to longer test execution times and maintenance. For this reason, it's important to perform continuous testing with the right application of tests, their prioritization, the environment required for them, and test maintenance.

8.2. Shift-Left and Shift-Right Testing

Shift left testing focuses on quality activities to be done at the beginning of the SDLC, that is, the requirement review, static analysis, unit testing and early security verification. The advantage with the use of this is the early detection of issues, which can possibly lessen rework and development cost [6, 15].

Shift-right testing involves moving quality activities into deployment and operation by monitoring, operational testing, resilience evaluation, and leveraging real usage feedback. It can be used to uncover failures which might be hard to reproduce in concerted test environments. But shift to the right must be consistent with the use of pre-release testing, as production environments can lead to potential user and organizational harm from undetected bugs.

8.3. AI-Assisted Software Testing

In addition to this, the domains for the application of software testing tasks like test-case generation, test prioritization, defect prediction, test maintenance, and failure analysis, are evaluated by means of AI and machine learning. In a recent mapping study of the software testing research, several research areas have been identified including machine learning, automated test generation, test evolution and test evolution automation techniques [6].

AI-powered testing can minimize human labor and aid in discovering testing opportunities within large and complex systems. But it is dependent on certain conditions, including availability of good-quality and representative data, reliability of the produced tests, and their capability to receive the results and verify them by AI. AI tools must be viewed as a tool to augment human expertise in testing, not replace it.

8.4. Critical Discussion

Our reviewed literature suggested that for all Software systems there is no single testing, testing level, testing tool and no single testing automation strategy that can be implemented. There is still a need for traditional testing levels as each type of testing (unit, integration, system, and acceptance) is a different type of evidence. Concurrently, with the modern way of development, these activities are being further automated, coupled with CI/CD and continuous feedback [4, 5].

While the benefits of automation are repeatability and speed, it adds the needs for maintenance and infrastructure. While preliminary testing can minimize rework at a lower stage in the system, it is still important to carry out system level and acceptance testing later or at the end in the system as some failures can only be seen when components are combined within the system or when they are tested under real world conditions by the users. AI-driven testing also provides avenues for further test generation and prioritization enhancements, though its results need to be verified and humanized.

The basic message here is that testing should be risk-oriented, have a sustained lifecycle, and be flexible. Evaluating the effectiveness of testing should also not only be based on the number of tests run, but on the evidence provided, the defects found, the quality attributes evaluated and the resources necessary to gather that evidence.

9. Conclusion, Limitations and Future Research

Software testing plays a vital part in software quality assurance and should not be limited to a last-minute verification activity, but should be done throughout the Software Development Life Cycle. The exam of this structured review involved an examination of role of testing throughout the SDLC, major testing levels, testing principles, the Software Testing Life Cycle, testing tools, automation, and some common testing practices. The analysis reveals that there is complementary evidence provided: Unit testing helps to build the confidence of early performance of the individual components; Integration testing helps identify defects related to interactions; System testing helps validate the overall system; and, Acceptance testing helps check if it's suitable for users and/or business requirements.

In addition, the review demonstrates that the traditional testing principles are still applicable to the present software engineering, however, they need to be adapted to iterative development, automation, continuous integration and DevOps environments. By automating and continuously testing, you can get immediate and consistent feedback, and shift-left and shift-right practices can enhance the journey of quality through the lifecycle. Concurrently, there are maintenance and infrastructure issues with the use of automation, and AI-generated testing must be properly validated and overseen by humans. It is therefore more understood that the essence of testing an effective product is a process which is risk-driven and lifecycle-oriented, than a set of separate testing activities.

9.1. Limitations

There are a few limitations to this review. First, it follows an approach to narrative with a structure that is different from a systematic review with statistical meta-analysis, so findings are qualitative synthesis, not quantitative estimates. Secondly, it has a review of technologies, tools and testing levels that are representative and not necessarily an exhaustive review of all testing technologies available. Third, AI supported testing, automation development and DevOps are new domains that could expand beyond the scope of this review. Lastly, the findings can only be generalized to software development in general because of the differences among various software domains and organizations with respect to software testing process.

9.2. Future Research

This review may be broadened into future studies in a few directions. Firstly, empirical research can evaluate the effectiveness of different types of testing in SDLC processes including Agile, DevOps, Scatter, and Sequential testing. Second, it is possible to study relationships between early testing, automation, early

defect detection and development cost across whole software projects in a longitudinal study. Third, the adoption of AI for test generation, test prioritization, and test maintenance could be scientifically assessed through systematic research and development activities, and help determine the areas in which AI offers tangible benefits over traditional methods. Another study might also look at comparison of testing procedures in a number of industries that have varying levels of risk, regulatory requirements and quality needs.

In conclusion, the software testing landscape promises to become more complex, exhibit increased automation, continuous testing, and use of smart testing methods as part of the testing lifecycle and quality assurance. Increase in technology, however, should not be without proper input from human minds, risk assessment and evidence-based evaluation for meaningful improvement and increased use of state-of-the-art technology in software testing.

Funding Statement: The authors received no external funding for this research.

Declaration of AI Use: AI tools were used for language editing and improving manuscript clarity. The authors verified all revisions.

Conflicts of Interest: The authors declare that they have no conflict of interest.

Data Availability: No new datasets were generated or analyzed in this study. The findings are based on published literature and publicly available software-testing standards and sources.

References

- [1] (Ed.). (2002). Information technology laboratory, technical accomplishments 2002. National Institute of Standards and Technology. <https://doi.org/10.6028/nist.ir.6909>
- [2] Zhao, Y., Serebrenik, A., Zhou, Y., Filkov, V., & Vasilescu, B. (2017). The impact of continuous integration on other software development practices: A large-scale empirical study. 2017 32nd IEEE/ACM International Conference on Automated Software Engineering (ASE), 60–71. <https://doi.org/10.1109/ase.2017.8115619>
- [3] Mishra, A., & Otaïwi, Z. (2020). DevOps and software quality: A systematic mapping. Computer Science Review, 38, 100308. <https://doi.org/10.1016/j.cosrev.2020.100308>
- [4] Soares, E., Sizilio, G., Santos, J., da Costa, D. A., & Kulesza, U. (2022). The effects of continuous integration on software development: a systematic literature review. Empirical Software Engineering, 27(3). <https://doi.org/10.1007/s10664-021-10114-1>
- [5] (Ed.). ISO/IEC/IEEE International Standard - Software and systems engineering --Software testing --Part 1:General concepts. IEEE. <https://doi.org/10.1109/ieeestd.2022.9698145>
- [6] (Ed.). ISO/IEC/IEEE International Standard - Software and systems engineering - Software testing -- Part 2: Test processes. IEEE. <https://doi.org/10.1109/ieeestd.2021.9591508>
- [7] Zampetti, F., Scalabrino, S., Oliveto, R., Canfora, G., & Di Penta, M. (2017). How Open Source Projects Use Static Code Analysis Tools in Continuous Integration Pipelines. 2017 IEEE/ACM 14th International Conference on Mining Software Repositories (MSR), 334–344. <https://doi.org/10.1109/msr.2017.2>
- [8] Stocco, A., Shehory, O., Jahangirova, G., Riccio, V., Barash, G., Farchi, E., & Saha, D. (2023). Software testing in the machine learning era. Empirical Software Engineering, 28(3). <https://doi.org/10.1007/s10664-023-10326-7>
- [9] Marir, T., Mokhati, F., & Bouchlaghem-Seridi, H. (2014). Do We Need Specific Quality Models for Multi-Agent Systems? - Toward Using the ISO/IEC 25010 Quality Model for MAS. Proceedings of the 9th International Conference on Software Engineering and Applications, 363–368. <https://doi.org/10.5220/0005097303630368>
- [10] Yan, M., Xia, X., Zhang, X., Xu, L., Yang, D., & Li, S. (2019). Software quality assessment model: a systematic mapping study. Science China Information Sciences, 62(9). <https://doi.org/10.1007/s11432-018-9608-3>
- [11] Yan, M., Xia, X., Zhang, X., Xu, L., Yang, D., & Li, S. (2019). Software quality assessment model: a systematic mapping study. Science China Information Sciences, 62(9). <https://doi.org/10.1007/s11432-018-9608-3>
- [12] Shahin, M., Ali Babar, M., & Zhu, L. (2017). Continuous Integration, Delivery and Deployment: A Systematic Review on Approaches, Tools, Challenges and Practices. IEEE Access, 5, 3909–3943. <https://doi.org/10.1109/access.2017.2685629>

- [13] Rasch, D. (1991). Pressman, R. S.: Software Engineering. Grundkurs für Praktiker. McGraw-Hill Software, Engineering. Hrsg.: Dipl.-Inform. John-Harry Wieken. McGraw-Hill Book Company GmbH, Hamburg 1989. 302 S., DM 63, 55, ISBN 3-89028-163-X. Biometrical Journal, 33(3), 378. Portico. <https://doi.org/10.1002/bimj.4710330333>
- [14] Sommerville, I. (1994). Software Engineering: Principles and Practice. Software Engineering Journal, 9(5), 228. <https://doi.org/10.1049/sej.1994.0029>
- [15] Daka, E., & Fraser, G. (2014). A Survey on Unit Testing Practices and Problems. 2014 IEEE 25th International Symposium on Software Reliability Engineering, 201–211. <https://doi.org/10.1109/issre.2014.11>
- [16] Yoo, S., & Harman, M. (2012). Regression testing minimization, selection and prioritization: a survey. Software Testing, Verification and Reliability, 22(2), 67–120. Portico. <https://doi.org/10.1002/stvr.430>
- [17] García, B., Gallego, M., Gortázar, F., & Munoz-Organero, M. (2020). A Survey of the Selenium Ecosystem. Electronics, 9(7), 1067. <https://doi.org/10.3390/electronics9071067>
- [18] Wang, Y., Mäntylä, M. V., Liu, Z., & Markkula, J. (2022). Test automation maturity improves product quality—Quantitative study of open source projects using continuous integration. Journal of Systems and Software, 188, 111259. <https://doi.org/10.1016/j.jss.2022.111259>
- [19] (Ed.). IEEE/ISO/IEC International Standard for Software and systems engineering--Software testing--Part 3: Test documentation. IEEE. <https://doi.org/10.1109/ieeestd.2021.9591577>
- [20] <https://doi.org/10.1109/ieeestd.2021.9591577>
- [21] Tahir, A., Rasheed, S., Dietrich, J., Hashemi, N., & Zhang, L. (2023). Test flakiness' causes, detection, impact and responses: A multivocal review. Journal of Systems and Software, 206, 111837. <https://doi.org/10.1016/j.jss.2023.111837>
- [22] Khaliq, Z., Farooq, S. U., & Khan, D. A. (2022). A deep learning-based automated framework for functional User Interface testing. Information and Software Technology, 150, 106969. <https://doi.org/10.1016/j.infsof.2022.106969>
- [23] Khatami, A., & Zaidman, A. (2024). State-of-the-practice in quality assurance in Java-based open source software development. Software: Practice and Experience, 54(8), 1408–1446. Portico. <https://doi.org/10.1002/spe.3321>
- [24] Ragkhitwetsagul, C., Krinke, J., Choetkiertikul, M., Sunetnanta, T., & Sarro, F. (2024). Adoption of automated software engineering tools and techniques in Thailand. Empirical Software Engineering, 29(4). <https://doi.org/10.1007/s10664-024-10472-6>
- [25] (Ed.). IEEE/ISO/IEC International Standard - Software and systems engineering--Software testing--Part 4: Test techniques. IEEE. <https://doi.org/10.1109/ieeestd.2021.9591574>