Alexander Vorvul

SOFTWARE QUALITY ASSURANCE FUNDAMENTALS

Table of Contents

I. Software Quality Assurance History

1. Software Quality Assurance History

Software quality assurance, as a discipline, has evolved over several decades, and its history can be divided into long-term periods that reflect significant changes in technology, methodologies, and practices.

These periods highlight the evolution of software testing from a manual, ad-hoc activity to a highly automated, AI-driven discipline.

Long-Term Periods

One may differentiate five important Software quality assurance periods prior to the end of the XXth century:

- 1947–1956 — Debugging period
- 1957–1978 — Demonstration period
- 1979–1982 — Destruction period
- 1983–1987 — Evaluation period
- 1988–2000 — Prevention period

Extending the trend for the XXIth century, one could single out the next three long-distance SQA testing eras:

- 2001–2011 — Test automation period
- 2012–2021 — Continuous testing period
- 2022–Present — AI-driven testing period

1.1. 1947–1956 Debugging period

The primary objective during the Debugging period revolved around detecting and resolving flaws, or "bugs", within software programs.

Unlike modern testing paradigms, which emphasize prevention and systematic validation, early efforts were predominantly centered on reactive debugging — locating and correcting errors after they manifested in the code.

Historical Context

This period corresponds to the formative years of computing — a time when electronic computers transitioned from theoretical constructs to practical tools capable of executing programmed instructions.

Software development was in its infancy, and structured approaches to quality assurance had yet to emerge.

Testing, as a formal discipline, did not exist.

Instead, the process was largely synonymous with debugging — an ad hoc, trial-and-error method of ensuring that programs functioned as intended.

Key Characteristics

The key characteristics of the Debugging period are as follows:

- **Developer-centric debugging** — Since specialized testing roles had not yet been established, programmers themselves were responsible for verifying their code.

  Testing was an intrinsic part of the development cycle rather than a distinct stage.

- **Absence of formalized tools and techniques** — Unlike today’s sophisticated testing frameworks, early debugging relied on manual inspection, print statements, and rudimentary diagnostic methods.

  There were no automated testing suites, standardized methodologies, or dedicated quality assurance teams.

- **Basic reliability as the primary goal** — The chief measure of success was whether a program executed without crashes or glaring inaccuracies in output.

  The concept of comprehensive validation, such as edge-case analysis, performance benchmarking, or user experience evaluation, was not yet a consideration.

Milestones

Below are the main milestones of the Debugging period:

- **Bug and debugging notions** — In 1947, the terms "bug" and "debugging" were coined when engineers working on the Harvard Mark II computer discovered an actual moth had got stuck in a relay, causing it not to make contact.

  Grace Murray Hopper detailed the incident in the work log, pasting the moth with tape as evidence and referring to the moth as the "bug" causing the error, and to the action of eliminating the error as "debugging".

- **Reactive Problem-Solving Mindset** — Debugging was perceived as a corrective measure rather than a preventive one.

  Developers addressed issues only after they arose, often in response to observable failures, rather than proactively designing tests to uncover hidden defects.

Debugging Period Role

This nascent stage of software testing laid the groundwork for future advancements in quality assurance.

At that time, the tests were focused on the hardware because it was not as developed as today and its reliability was essential for the proper functioning of the software.

While primitive by contemporary standards, the emphasis on debugging established the foundational principle that software must be scrutinized for errors — a concept that would later evolve into systematic testing methodologies, specialized tools, and dedicated QA professions.

1.2. 1957-1978 Demonstration Period

The central objective of the Demonstration period shifted toward proving that the software operated in strict accordance with its intended design and specifications.

Unlike the earlier era, where testing was synonymous with reactive debugging, this period emphasized validation, ensuring that the software not only ran without errors but also fulfilled its predefined functional requirements.

Historical Context

As software systems expanded in scale, functionality, and criticality, the limitations of ad-hoc debugging became increasingly apparent.

Organizations recognized that merely eliminating crashes was insufficient — software now needed to meet business and operational needs for sure.

This era saw the gradual transition from informal, developer-led error-fixing to a more structured — though still nascent — effort to verify software correctness.

Testing remained relatively informal compared to modern standards, but its purpose evolved — shifting from defect correction to requirement validation.

The focus was no longer solely on "making the program work" but on demonstrating that it worked as expected under defined conditions.

Key Characteristics

The key characteristics of the Demonstration period are as follows:

- **Validation-centric approach** — Testing was primarily employed to affirm that the software executed its intended functions correctly, rather than aggressively uncovering hidden flaws.

  The mindset leaned toward confirmation — "Does it work as specified?" — rather than investigation — "Where does it fail?".

- **Defect-averse mindset** — The emphasis was on proving the absence of critical defects, often leading to optimistic assessments.

  Unlike later methodologies that actively sought weaknesses — for instance, stress testing, boundary analysis, etc — this approach assumed correctness unless evidence proved otherwise.

- **Late-stage testing** – Testing was frequently conducted toward the end of development, often as a final checkpoint before release.

  This "test-last" mentality contrasted with modern iterative testing, where evaluation occurs continuously throughout the development lifecycle.

- **Manual and Ad-Hoc execution** — Despite the growing need for validation, testing lacked standardized methodologies or automation.

  Most checks were performed manually, relying on developer intuition or rudimentary test cases.

Milestones

- **Tests development** — In 1957, **Charles Baker** explained the necessity for the test development aimed at ensuring the software meets its pre-designed requirements.

  Thus, the distinction between software functionality control — Debugging — and software quality maintenance — Testing — was introduced for the Testing to be carried out as a separate activity.

- **Tests importance** — Test development became more important as more expensive and complex applications were being developed, and the cost of solving all these deficiencies affected a clear risk to the profitability of the project.

  A special focus was placed on increasing the quantity and quality of tests. For the first time the quality of an application began to be linked to the state of the testing stage.

  The goal of testing was to demonstrate that the software performed what it had initially been designated for, using expected and recognizable parameters.

- **Emergence of Verification and Validation** — V&V — The distinction between verification — "Are we building the product right?" — and validation — "Are we building the right product?" — began to take shape.

  This framework laid the groundwork for later quality assurance disciplines.

- **Continued reliance on manual processes** — While the philosophy of testing matured, practices remained labor-intensive and unstructured.

The absence of systematic test design — e.g., test plans, coverage metrics — meant that effectiveness varied widely across projects.

Demonstration Period Role

The Demonstration period represented a critical transition moving software quality assurance from **reactive debugging** toward **deliberate validation**.

However, the lack of formalized processes and early testing integration meant that many latent defects still escaped detection until deployment.

These gaps would later drive the development of methodical testing frameworks, automated tools, and proactive quality measures to bridge the debugging and modern testing in subsequent eras.

1.3. 1979-1982 Destruction Period

The Destruction period represented a fundamental transformation in software quality assurance approach, where the primary objective evolved from proving correctness to actively seeking and exposing defects.

Testing was no longer viewed merely as a validation step but rather as a systematic effort to stress, challenge, and intentionally break the software under controlled conditions.

The underlying premise was that improving software quality required aggressively identifying weaknesses before release, rather than passively confirming functionality.

Historical Context

As software systems grew more sophisticated and became more mission-critical, the limitations of traditional verification-focused testing became apparent.

A proactive defect-hunting mindset emerged, recognizing that:

- Software reliability could not be assumed — it had to be rigorously challenged
- The absence of observed failures did not equate to correctness — flaws often lay hidden in untested scenarios
- Human psychology influenced testing effectiveness — developers naturally avoided tests that might "break" their code

The Destruction period marked the professionalization of testing as a distinct discipline, shifting from an optimistic demonstration of functionality to a pessimistic, investigative process aimed at uncovering faults.

Key Characteristics

Below are the key characteristics of Destruction period:

- **Formalization of testing** — Testing transitioned from an informal, post-development activity to a structured stage integrated into the software development lifecycle.

  Dedicated test cases, plans, and documentation became standard, ensuring systematic evaluation rather than ad-hoc checks.

- Methodical testing techniques —- Some techniques represented early efforts to systematize test design, moving beyond trial-and-error were initially introduced during the Destruction period:
  - **Boundary value analysis** — BVA — Focused on testing at the edges of input ranges, where defects frequently cluster
  - **Equivalence partitioning** — Reduced redundancy by grouping inputs that should trigger similar behaviors, optimizing test coverage

- **Defect maximization objective** — The success of testing was measured by its ability to expose flaws, not just confirm expected behavior.

  This required creative, adversarial thinking — crafting scenarios that exploited potential weaknesses in logic, input handling, or edge cases.

- **Proactive mindset** — Testing was no longer about waiting for failures to emerge in production but preemptively forcing failures in development.

  This shift reduced the cost and risk of late-stage defect discovery.

Milestones

- In 1979, **Glenford Myers**, with the definition below, radically redefined the procedure for detecting faults in the program:

  *"**Software testing** is the process of running a program with the intention of finding errors."*

  Myers' concern was that in pursuing the goal of demonstrating that a program is flawless, one could subconsciously select test data that has a low probability of causing program failures, whereas if the goal is to demonstrate that a program is flawed, our test data will have a greater probability of detecting them and we will be more successful in testing and thus in software quality.

  From now on, the tests will try to demonstrate that a program **does not work as it should**, contrary to how it was done until then.

  This reorientation laid the groundwork for modern testing theory, prioritizing **defect detection** over **confirmation**, leading to new techniques of testing and analysis.

- **Destructive testing methodologies** — Myers' philosophy spurred the development of new techniques explicitly designed to:

  - Expose hidden assumptions in code
  - Challenge error-handling robustness
  - Reveal edge-case vulnerabilities

  Testing was no longer a passive checkpoint but an active quality-improvement mechanism.

Destruction Period Role

The Destruction period permanently altered software engineering practices by establishing core principles that endure today:

- **Testing must be skeptical —** assuming defects exist until proven otherwise
- **Quality is achieved through rigorous challenge** — not just affirmation
- **Test design is a distinct skill** — requiring specialized knowledge beyond coding proficiency

The introduction of structured techniques — BVA, Equivalence partitioning, etc — and Myers' psychological insights formed the foundation for subsequent advancements in automated testing, risk-based testing, and continuous quality assurance.

1.4. 1983-1987 Evaluation period

The Evaluation period represented a fundamental transformation in how the software industry conceptualized and implemented quality control.

Rather than treating testing as a final gatekeeping activity, organizations began embracing a comprehensive philosophy of continuous quality evaluation spanning the entire software development life cycle.

This paradigm shift moved quality considerations:

- From being **reactive** to being **proactive**
- From being **fragmented** to being **integrated**
- From being **defect-focused** to being **quality-centric**

Historical Context

By the early 1980s, several converging factors necessitated this evolution:

- Increasing software complexity in business and mission-critical systems
- Rising costs associated with post-release defect remediation
- Growing recognition that quality cannot be "tested in" but must be built into the process
- Maturing understanding of software engineering as a disciplined practice

The software industry transitioned its view on testing:

- from **separate stage** to an **integrated process**
- from **defect detection** tool to a **quality assurance** mechanism
- from **final verification** step to a **continuous evaluation** practice

Key Characteristics

The key characteristics of the Evaluation period are as follows:

- **Lifecycle-wide quality assessment** — Quality evaluation became embedded in every Software development life cycle stage:
    - Requirements analysis — testability verification
    - Design — architectural reviews
    - Development — unit testing
    - Deployment — system testing
    - Maintenance — regression testing

  The development gained a V-shaped prominence, explicitly mapping test activities to each development stage.

- **Formalization of testing practices** — Standardized methodologies emerged — for example, IEEE 829:
    - Documentation — test plans, cases, procedures — became mandatory
    - Metrics were instituted and measurement programs were developed
    - Specialized QA roles and organizational structures developed

- **Quality assurance as an organizational function** — Quality Assurance became distinct from Testing, encompassing:
    - Process definition and improvement
    - Prevention-oriented activities
    - Organizational quality culture
    - Metrics and benchmarking

  The **Quality Gate** concept institutionalized checkpoints throughout the software development lifecycle where specific quality criteria must be met before a project can progress to the next stage.

- **Process-driven testing approach**:
    - Repeatable, defined test processes replaced Ad-hoc methods
    - Test planning became forward-looking rather than reactive
    - Traceability matrices linked requirements to test cases
    - Formal entry and exit criteria governed test stages

Milestones

- In 1983, IEEE 829 **Standard for Software Test Documentation** established uniform documentation requirements including:
    - Test plan structure
    - Test case specifications
    - Test procedure definitions
    - Test log requirements
    - Incident reporting formats
    - Test summary reporting

  The Standard introduced a common language for test professionals enabling process consistency across organizations.

- The advent of **Automated testing tools** was another significant achievement of the Evaluation period when:
    - Early test automation frameworks emerged
    - Regression testing became practical at scale
    - The foundation for modern continuous testing was laid
- Institutionalization of **Quality Metrics** introduced:
    - Defect density measurements
    - Test coverage percentages
    - Requirements traceability matrix
    - Escape defect analysis

  These provided quantitative quality assessments.

- Legacy and Transition to **Modern Practices** — the Evaluation Period established foundational concepts that continue to shape quality practices today:
    - The principle that quality is a process, not an event
    - The understanding that testing must be planned and managed
    - The recognition that quality requires organizational commitment
    - The framework for integrating testing throughout development

Evaluation Period Role

This era's emphasis on standardization, documentation, and process orientation directly enabled subsequent innovations like:

- Capability Maturity Model — CMM — integration
- Agile testing methodologies
- Continuous integration/continuous delivery — CI/CD — pipelines
- DevOps quality practices

The institutionalization of quality assurance during the Evaluation period transformed software testing from a technical afterthought to a professional engineering discipline, establishing patterns and practices that remain relevant for decades later.

1.5. 1988-2000 Prevention period

The Prevention period marked a paradigm shift in software quality management, transitioning from reactive defect detection to proactive defect prevention.

The industry recognized that finding bugs late in the software development cycle was costly and inefficient, leading to a strategic emphasis on early testing, rigorous process controls, and continuous improvement.

Testing evolved from being a post-development checkpoint to an integrated, iterative practice embedded throughout the software development lifecycle — SDLC.

The goal was no longer just to identify flaws but to prevent them from occurring in the first place through systematic process enhancements and early validation.

Historical Context

Several key factors drove this transformation:

- **Increasing Software Complexity** — As applications grew larger and more interconnected, late-stage defect resolution became prohibitively expensive
- **Cost of Late Defects** — Studies showed that bugs found after the software release were 10–100 times more expensive to fix than those caught early
- **Demand for Faster Releases** — The emergence of internet-driven business models necessitated shorter development cycles, making traditional "test-last" approaches obsolete
- **Maturity of Software Engineering** — The discipline began adopting formalized best practices inspired by manufacturing quality control

This period saw the professionalization of testing as a strategic function, rather than a tactical afterthought.

Key Characteristics

The key characteristics of the Prevention period are as follows:

- **Early and Continuous Testing** — or Shift-Left — Testing activities moved as early as possible in the SDLC with the relevant practices introduced:
    - **Requirements reviews** – Formal inspections to ensure clarity, completeness, and testability before coding began
    - **Design inspections** – Rigorous evaluation of architectural decisions for potential flaws
    - **Code walkthroughs and Peer reviews** – Collaborative defect prevention via manual code analysis

  The practices led to reduced rework by catching ambiguities and design flaws before implementation.

- **Process-centric quality assurance:**
    - **Quality Gates** — Mandatory checkpoints — at the end of each stage and alike — to enforce standards
    - **Standardized methodologies** — Adoption of frameworks like Capability Maturity Model and Agile Iterative Development which enabled continuous feedback via incremental testing
    - **Metrics-driven improvement** – Defect density, escape rates, and test coverage became key performance indicators

- **Tooling and Automation advancements:**
    - **Static analysis tools** — Automated code review for early defect detection, for instance, linting tools
    - **Test automation frameworks** — Dedicated software testing automation and software quality assurance products emerged

    - **CI/CD precursors** — Early Continuous Integration and Continuous Development tools enabled frequent validation

- **Cultural shift toward quality ownership:**
    - **Shared responsibility** — Developers, testers, and business analysts collaborated on quality
    - **Prevention mindset** — Teams focused on getting it right the first time rather than relying on downstream testing

Milestones

- **Formalization of Shift-Left testing —** Testing early and often principle became a cornerstone of modern SDLCs and influenced later methodologies like Continuous Testing
- **Rise of Agile iterative development:**
    - **Scrum** — 1995 — and **Extreme Programming** — XP, 1996 — methodologies embedded testing into short cycles
    - **Test-Driven Development** — TDD — emerged as a prevention-focused practice
- **Growth of Testing tools and frameworks**
- **Industry Standards Expansion:**
    - **ISO 9000-3** — 1997 — Provided QA guidelines for software development
    - **IEEE 1012** — 1998 — Standardized verification and validation processes

Prevention Period Role

The Prevention Period laid the groundwork for the future software quality assurance process enhancements:

- **DevOps** — Merging development and operations with quality as a shared goal

- **Shift-Right** — Complementing early testing with production monitoring
- **Quality Engineering** — Beyond "testing", encompassing design-for-quality principles

This period's emphasis on prevention, automation, and process discipline remains central to contemporary software engineering, proving that proactive quality assurance is far more effective than reactive debugging.

1.6. 2001-2011 Test Automation Period

The first decade of the XXIth century marked a transformative period in software quality assurance, characterized by the widespread adoption of automation and the emergence of new testing paradigms.

As software systems grew in complexity and release cycles accelerated, manual testing became increasingly impractical.

This period saw the industrialization of testing through automation frameworks, agile-aligned methodologies, and early AI-assisted tools — laying the foundation for modern DevOps and continuous testing practices.

Historical Context

Several key drivers fueled the shift to systematic and intelligent testing focused on efficiency and scalability:

- **Explosion of Web applications** — The dot-com boom and Web 2.0 demanded cross-browser compatibility and 24/7 reliability
- **Agile adoption** — Shorter development cycles required faster feedback loops via automation
- **Cost Pressure** — Studies showed automation could reduce regression testing effort by more than a half
- **API-Centric Architectures** — Service-Oriented Architecture — SOA — made **API testing** a critical need

Key Characteristics

Below are the key innovations and methodologies of the Automation era:

- **Test-Driven Development** — TDD — Writing tests **before** code greatly reduced defect rates and forced modular, testable software design
- **Behavior-Driven Development** — BDD — Bridged business-IT gaps using natural-language specifications using human-readable syntax, aligned testing with user stories, and Enabled non-technical stakeholders to validate logic
- **The Selenium revolution** — 2004 — Selenium WebDriver, which solved browser automation with cross-browser support and language-agnostic bindings became the de facto standard for web UI testing
- **API testing maturity** — The shift from UI-centric to API validation enabled early — shifted left — integration testing and performance benchmarking
- **Cloud-based testing platforms** — Eliminated lab maintenance costs and provided real-device testing, which is critical for mobile development

Automation Period Role

The Automation period transformed testing from a manual job to a strategic, tech-driven discipline — proving that Quality at Speed was achievable through innovation.

1.7. 2012-2021 Continuous Testing Period

The period between 2012 and 2021 marked a revolutionary shift in software development and testing, driven by the widespread adoption of DevOps principles and Continuous Testing.

This era was defined by the convergence of development — **Dev**, testing — **QA**, and operations — **Ops**, transforming traditionally isolated functions into a collaborative, automated pipeline.

The primary goal was to accelerate software delivery while maintaining high quality, achieved through automation, infrastructure innovation, and cultural change.

Historical Context

Several industry trends necessitated this evolution:

- **Demand for faster releases** — The rise of cloud computing and Software as a Service models required continuous deployment, that is daily or hourly releases
- **Microservices architecture** — Distributed systems increased complexity, making End-to-End testing more challenging
- **Agile at scale** — Large organizations adopted Scrum, Kanban, and Scrum agile framework — SAFe, — requiring testing to keep pace with rapid iterations
- **Cost of manual processes** — Manual testing bottlenecks appeared to significantly delay releases.

This period saw the death of the "throw-it-over-the-wall" mentality, replacing it with shared ownership of quality.

Key Characteristics

- **Shift-Left**, or Early Testing**:**
    - Testing moved earlier in the SDLC, with developers writing Unit tests, Integration tests, and API tests alongside code
    - Test-Driven Development, TDD, and Behavior-Driven Development, BDD, became standard in Agile teams
- **Shift-Right**, or Production Testing, extended Testing into production via Real-user monitoring — or RUM, — A/B Testing, and other techniques
- **Infrastructure as Code** — IaC:
    - **Containerization tools** — such as Docker, 2013 — revolutionized testing by:
        - Packaging apps and dependencies into lightweight, portable containers
        - Enabling consistent test environments, eliminating "works on my machine" issues
    - **Orchestration tools** — such as Kubernetes, 2014 — automated container governance, allowing scalable test clusters
    - **IaC Tools** — such as Terraform, Ansible — automated environment provisioning, reducing setup time from days to minutes
- **CI/CD pipelines and Automated testing** — Continuous integration and Continuous development tools allowed automating:
    - local and in-cloud Build-Test-Deploy cycles
    - parallel test execution across environments

- **Site Reliability Engineering** — SRE — emerged, blending Operations and Quality Assurance.

Continuous Testing Period Role

This period democratized testing, making it every engineer’s responsibility rather than a QA-only task.

It transformed testing from a slow, manual process to a fast, automated, and intelligent practice.

By integrating testing into CI/CD, infrastructure, and monitoring, it laid the groundwork for today’s autonomous, AI-enhanced practices giving the birth for **Continuous Quality**.

Continuous testing became a necessity rather than an option, ensuring that software kept pace with the demands for rapid delivery and high user expectations.

It proved that speed and quality are not trade offs, but mutually achievable goals.

1.8. 2022-Present AI-Driven Testing

The current testing period, Artificial Intelligence — AI — and Machine Learning — ML — are fundamentally redefining how testing is conceived, executed, and optimized.

AI-driven testing — or AI-powered testing — is the use of Artificial Intelligence and Machine Learning to automate and enhance software testing processes.

No longer confined to scripted validations, testing has evolved into an intelligent, predictive, and self-improving process that spans the entire software lifecycle.

Historical Context

Several converging trends have accelerated this transformation:

- **AI maturation** — Breakthroughs in deep learning enabled practical applications beyond research labs
- **Testing complexity crisis** — With systems becoming more complex, traditional methods became economically unsustainable
- **Generative AI explosion** — The 2022 ChatGPT release proved AI's potential to create test artifacts, not just analyze them

Key Characteristics

Key Technologies Behind AI-Driven Testing are:

- **Machine Learning** — ML — Improves test scripts over time by learning from past executions

- **Natural Language Processing** — NLP — Converts plain-text requirements into automated test cases
- **Computer Vision** — Uses image recognition for UI testing, for instance, identifying dynamic elements
- **Predictive Analytics** — Identifies high-risk areas needing more testing

Milestones

The key characteristics of the AI-Driven Testing are as follows:

- **Self-Healing Test Automation** — Traditional tests broke due to:
    - Dynamic UIs
    - Environment inconsistencies
    - Flaky network conditions

  AI Tools use ML to:

    - Detect when element locators change
    - Automatically update selectors while maintaining test intent
    - Learn from corrections to improve future resilience

This reduces test maintenance, enabling sustainable automation at scale.

- **Visual validation with Computer vision:**
    - Traditional tools validated HTML structure
    - Visual AI tools use convolutional neural networks — CNNs — to detect visual regressions at pixel level and ignore non-impactful changes
- **ML-powered test impact analysis** — ML tools can:
    - Analyze code changes via static analysis
    - Map modifications to affected test cases

    - Prioritize test execution based on historical defect data, code complexity metrics, and business criticality

- **Codeless automation** tools lower the automation barrier the following ways:
    - Use Natural language processing — NLP — to translate plain languages to test scripts
    - Provide visual modeling interfaces
    - Auto-generate maintenance scripts

  As a result, codeless automation:
    - Enables subject matter experts to create tests
    - Reduces automation skills gap
    - Accelerates test coverage expansion

AI-Driven Testing Period Role

The current era does not just change how we test, but what it means to deliver quality software in an AI-driven world.

The organizations thriving in this new paradigm are those treating quality as a strategic differentiator rather than a compliance checkpoint.

Unlike traditional scripted automation, AI-driven testing leverages intelligent algorithms to:

- Generate test cases
- Self-heal test scripts
- Predict defects
- Optimize test coverage
- Analyze test results

As a result, the most obvious benefits of AI-Driven Testing are:

- **Faster Test Creation** — AI generates tests from requirements or user behavior

- **Reduced Maintenance** — Self-healing tests minimize script updates
- **Smarter Test Execution** — Prioritizes critical test cases
- **Improved Accuracy** — Reduces false positives and negatives
- **Enhanced Test Coverage** — AI explores edge cases humans might miss

While not a complete replacement for human testers, AI significantly reduces manual effort and improves software reliability.

AI will not replace testers in the nearest future.

But testers who use AI may definitely replace those who don’t.

II. Software Quality

2. Software Quality

Quality

Quality is a set of inherent properties of an object that allows it to satisfy implicit or explicit needs.

Software quality

Software quality is the degree to which a software reliably performs its intended operations without errors or deviations from specified requirements.

It should mean that the application is free from vulnerabilities, either intentionally designed into the software or accidentally inserted at any time during its lifecycle.

Assurance

Assurance is a positive declaration on a product or service, which gives confidence.

It provides a guarantee that the product works correctly as per the expectations or requirements.

Quality Assurance

Quality assurance — QA — is a way of preventing defects in created products and avoiding problems when delivering products or services to customers.

Quality assurance focuses on improving the development process and making it efficient and effective as per the quality standards defined for products.

Software quality assurance

Software quality assurance — SQA, or QA — is the constant coordination of the engineering processes aimed to safeguard proper quality of the software and its compliance against the defined standards.

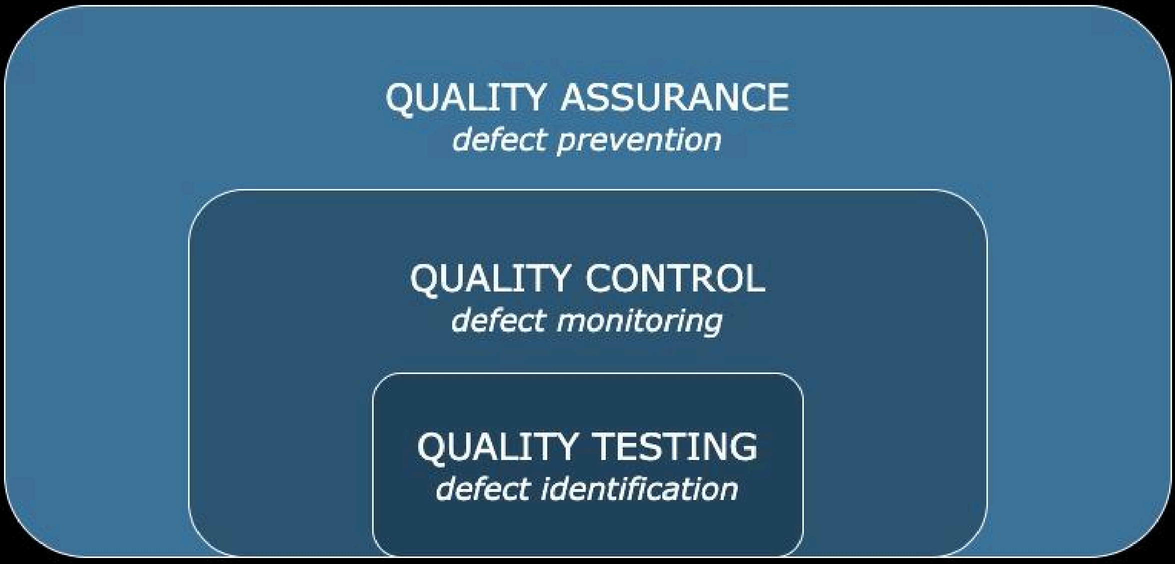
Software quality assurance engineer

Software quality assurance engineer — SQAE — is a specialist focused on ensuring the software quality against the requirements and standards during all stages of the software development process.

2.1. Software Quality Scopes

Software quality encompasses three critical scopes, each playing a distinct role in delivering reliable products:

- **Software quality testing** is the activity aimed at detecting issues in the product and executing systematic checks to identify defects, covering functional, performance, and security validations
- **Software quality control** is a terminal process of issues detection in a product before it is delivered to end users which involves defect scanning through inspections, reviews, and real-time monitoring
- **Software quality assurance** is a process which assures that all software engineering processes, methods, activities, and work items are monitored, streamlined and comply with the defined standards



Software quality scopes

Together, these scopes form a holistic quality framework ensuring software meets both technical requirements and user expectations efficiently:

- **Testing** identifies defects and verifies correctness
- **Control** monitors quality and enforces compliance
- **Assurance** prevents defects and improves processes

By integrating all three, organizations achieve higher reliability, reduced costs, and sustained customer trust.

2.2. Software Quality Testing

Software Quality Testing is an evaluation-oriented activity designed to ensure a software application meets specified requirements, functions correctly, and delivers a high-quality user experience.

It involves identifying defects, verifying functionality, and assessing performance, security, usability, and reliability.

Testing can be performed by a tester or a dedicated team of testers.

This process may also include a stage of test planning.

Key Objectives

Software quality testing is driven by several key objectives that collectively ensure the delivery of a robust and reliable product.

These objectives include:

- **Defect detection** — to systematically identify and document bugs, errors, and inconsistencies prior to release
- **Requirements validation** — to verify that the final software product aligns with all defined business objectives and user requirements
- **Performance evaluation** — to assess critical non-functional attributes such as speed, scalability, and stability under expected load conditions
- **Security assurance** — to proactively identify security vulnerabilities and ensure the software is protected against potential threats and breaches

- **Usability verification** — to evaluate the user interface and experience, ensuring the software is intuitive, efficient, and satisfactory for the end-user
- **Compliance verification** — to confirm that the software adheres to all applicable industry standards, legal regulations, and internal policies

Software Testing Role

The primary purpose of testing is to detect software failures so that defects may be discovered and corrected.

Software quality testing:

- **Reduces risks** of software failures
- **Improves experience** of the customer
- **Lowers costs** of the long-term maintenance
- **Ensures compliance** with industry regulations

The role of software testing often means the examination of code as well as its execution in various environments and conditions being guided by the following statements:

- Testing **cannot** establish that a product functions properly under all conditions
- Testing **can only** establish that it does not function properly under specific conditions.

By implementing rigorous testing processes, organizations can deliver reliable, secure, and high-performing software products.

2.3. Software Quality Control

Software Quality Control — SQC — is a verification-oriented activity designed to identify and correct software defects by verifying adherence to predefined quality standards.

SQC focuses on detecting deviations from requirements through systematic testing, reviews, and inspections before the product reaches users.

Key Objectives

The main Quality Control objectives are as follows:

- **Defect detection** — Finding bugs, errors, or non-conformities in code, design, or documentation
- **Standards compliance** — Ensuring alignment with functional requirements, industry standards, and organizational guidelines
- **Process enforcement** — Validation the development workflows, including coding practices and testing protocols, are followed

Key Activities

Software Quality Control activities usually include:

- **Testing** — Execute test cases to uncover functional or non-functional defects
- **Peer Reviews** — Manual examination of code or documents by team members
- **Static Analysis** — Automated checks for code quality issues without execution
- **Dynamic Analysis** — Software behavior monitoring during runtime
- **Audits** — Formal inspections to verify compliance with processes and standards

Quality Control Role

During the software quality control, testing team verifies the product's compliance with the functional requirements and this way:

- **Reduces post-release failures** — crashes, security breaches, etc
- **Saves costs** — avoiding the defects post-launch correction which is many times more expensive
- **Builds user trust** — by delivering reliable, high-quality software

2.4. Software Quality Assurance

Software Quality Assurance

Software quality assurance — SQA, or QA — is a planned and integral validation-oriented activity focused on meeting requirements for a product's quality aimed at further quality system improvement.

SQA assures that all software engineering processes, methods, activities, and work items are monitored, streamlined, and comply with the defined standards.

Software Quality Assurance incorporates all software development processes starting from defining requirements to coding until release.

Key Role

Software Quality Assurance is based on engineering processes that guarantee quality in a more efficient way than Software Quality Control.

Software Quality Testing and Software Quality Control are able to detect the major amount of issues in the product but this doesn't mean that these defects won't take place again.

The role of SQA is to re-engineer the system so that further occurrence of these defects won't happen; in fact, it may not include any testing at all.

The prime goal of the Software Quality Assurance is to ensure the quality of software products or services provided to the customers by an organization.

Quality Control and Quality Assurance

Software Quality Control is distinct from Software Quality Assurance.

Software Quality Control is a validation of artifacts' compliance against established criteria — **finding defects**.

Software Quality Assurance encompasses processes and standards for ongoing maintenance of high quality products, for instance, software deliverables, documentation, and processes — **avoiding defects**.

The difference between Software Quality Control and Software Quality Assurance may be stated as follows:

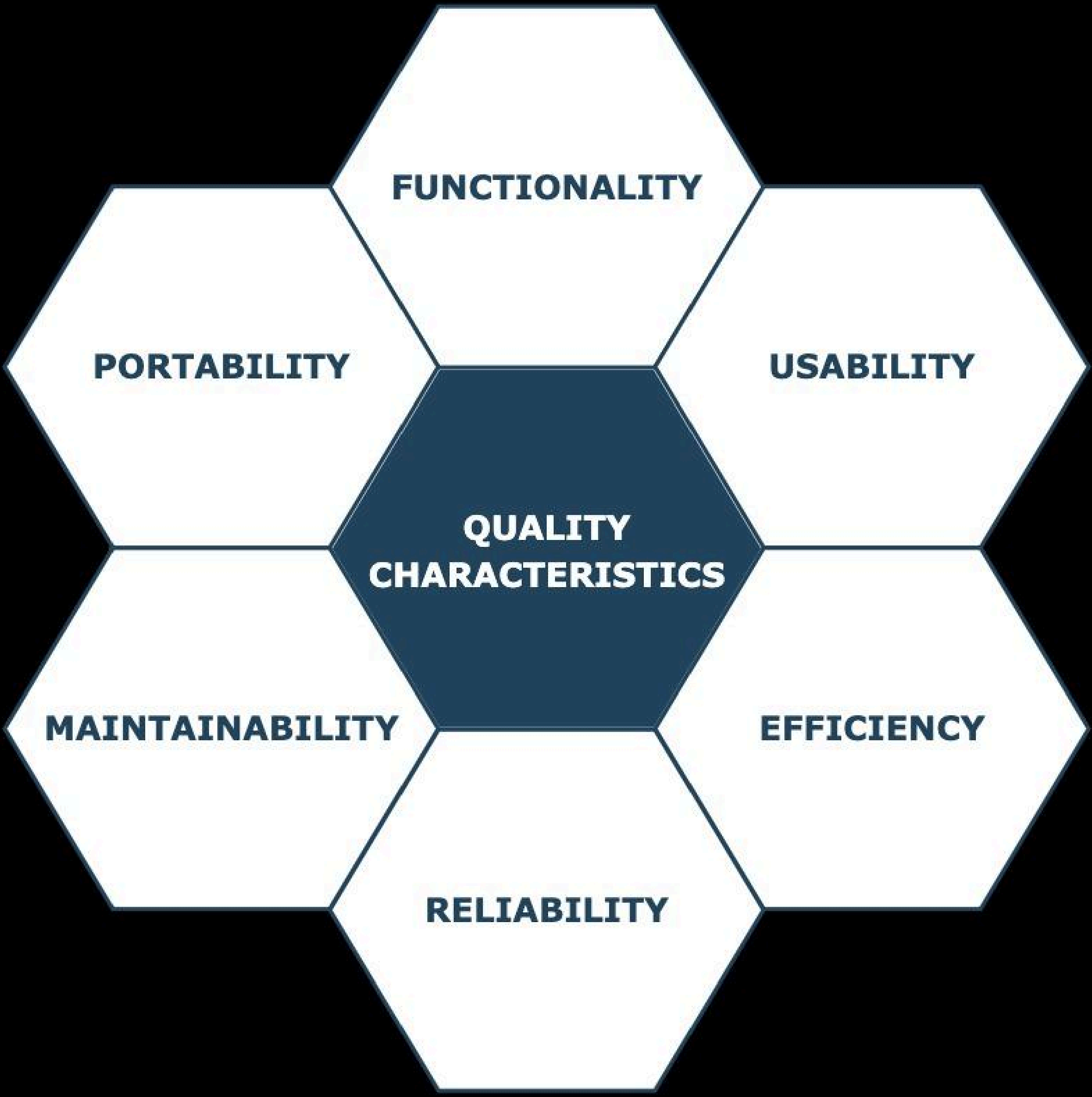
- SQC is a **product** oriented activity
- SQA is a **process** oriented activity

Software Quality Control keeps track of software development **results** to comply with requirements.

Software Quality Assurance traces all the software development **processes** to follow in due manner.

2.5. Software Quality Characteristics

Software quality characteristics — often called Quality Attributes or Non-Functional Requirements — are the measurable properties of a software system that define its fitness for the purpose, customer experience, and adherence to requirements.



The main software quality characteristics

These characteristics serve as benchmarks to evaluate how well the software performs, behaves, and meets stakeholder expectations.

Functionality

Functionality is the ability of a software to bear specified properties.

Functionality embraces:

- Suitability
- Integrity
- Security
- Accuracy, and other properties

Usability

Usability is the capability of a software to be easily understood and applied.

Usability comes over:

- Operability
- Learnability
- Accessibility
- Attractiveness, and so on

Efficiency

Efficiency is the relationship between the level of software performance and the amount of resources needed.

Efficiency involves:

- Capacity
- Time behavior
- Resource utilization

- Effectiveness, and so forth

Reliability

Reliability is the ability of software to continue functioning under stated conditions over a given period of time.

Reliability spans:

- Recoverability
- Availability
- Stability
- Consistency, etc

Maintainability

Maintainability is the effort needed to make specified modifications.

Maintainability takes in:

- Scalability
- Reusability
- Modularity
- Testability, and others

Portability

Portability is the ability of the software to be transferred from one environment to another.

Portability reaches out to:

- Installability
- Replaceability
- Compatibility
- Coexistence, etc

Key Role

Understanding and prioritizing software quality characteristics is a fundamental activity in software engineering.

2.6. Entry and Exit Criteria

Criterion is a principle or standard to judge something.

Each stage of the Software development and Software testing life cycle has its own Entry and Exit criteria.

Entry Criteria

Entry criteria are the criteria which must be met before initiating a specific development or testing stage.

It is a predefined set of conditions that must exist before a unit of development or testing work can commence.

It is used as a process control mechanism to determine the cost-effectiveness of initiating stage activities.

The team should enter the next stage only after the exit criteria for the previous one is met.

Exit Criteria

Exit criteria are the criteria which must be met before completing a specific development or testing task or a process.

It is a predefined set of conditions that must exist before a unit of development or testing work can be deemed completed.

It is used as a process control mechanism to verify that a development or testing stage has been completed and that its products are of acceptable quality.

Exit criteria define the deliverables to be completed before the stage to be left.

III. Software Development Life Cycle

3. Software Development Life Cycle

Software Development Life Cycle — SDLC — is a process of software production with intent to design, develop and test an application of the highest quality, at lowest cost, and in the shortest time possible.

SDLC — also called **Software development process**, SDP — may be named a framework to define tasks performed at each step of the software production.

SDLC Stages

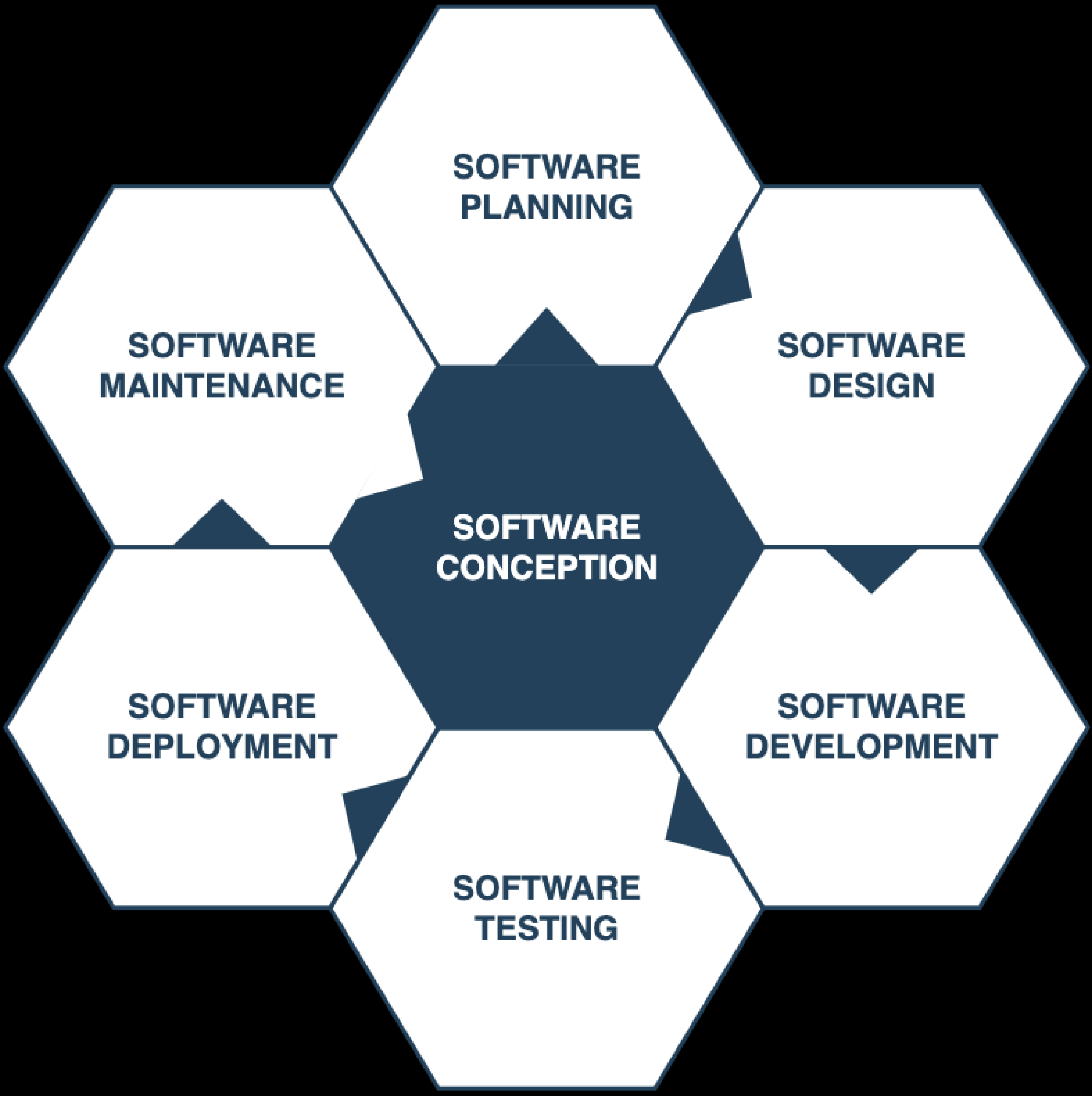
Software development efforts are structured into several stages and the Software Development Life Cycle serves to encompass them.

The Cycle does not conclude until all the requirements have been fulfilled, and will continue until all the potential needs are adjusted within the system.

SDLC provides a well-structured flow of stages that help an organization to quickly produce high-quality, well-tested, and ready for production use software.

Software Development Life Cycle consists of a detailed consequence of steps describing how to develop, maintain, alter, and replace specific software, and includes the following stages:

1. Software Conception
2. Software Planning
3. Software Design
4. Software Development
5. Software Testing
6. Software Deployment
7. Software Maintenance



Software Development Life Cycle

Benefits

SDLC works by lowering the cost of software development while simultaneously improving quality and shortening production time.

SDLC achieves these apparently divergent goals by following a plan that removes the typical pitfalls of software development projects.

SDLC has a strong focus on the testing stage — being a repetitive methodology, it ensures code quality at every cycle.

The biggest advantage of the Software Development Life Cycle is that it provides control of the development process to a certain extent and ensures the system complies with all the requirements that have been specified.

Drawbacks

SDLC does not work so well where there are levels of uncertainty or unnecessary overheads.

It directs the development efforts with an emphasis on planning, but the system does not encourage creative input or innovation throughout the lifecycle.

3.1. Software Conception Stage

Software Conception is the stage where end user business requirements are analyzed and project goals converted into defined system functions that the organization intends to develop.

That's the reason for it to be also called **Software Ideation** or **Requirements Analysis** stage.

Requirements Analysis

Conducted during the Conception stage, Requirements Analysis is the foundation of successful software development as it ensures the final product aligns with business goals, user needs, and technical feasibility.

The Requirements Analysis is indispensable for the following main reasons:

- Defines clear project scope:
  - prevents scope creep avoiding uncontrolled feature additions
  - sets priorities distinguishing must-have and nice-to-have features
- Aligns stakeholders:
  - bridges gaps between developers, clients, and end-users of the software
  - reduces miscommunication with documented requirements serving as a single source of truth
- Identifies risks early:
  - technical feasibility uncovers impractical demands
  - regulatory compliance flags legal needs

- Saves time and cost:
    - fixes during conception cost much less than fixes in development and many times more than post-release fixes
- Drives design and testing:
    - guides architecture determining whether to use microservices, monoliths, or serverless
    - forms test cases directly from requirements
- Ensures user-centric outcomes:
    - user stories or use cases capture real-world workflows
    - avoids useless features which are rarely or never used

Key Activities

The Requirements Analysis is performed by the senior members of the team with inputs from the customer, sales department, market surveys and domain experts in the industry.

The Requirements Analysis and elicitation consists of the following primary activities:

- Stakeholder Interviews — Input gathering from clients, users, and business teams
- Market Research — Competitors and industry standards analysis

Entry and Exit Criteria

The main triggering events for the software conception include:

- Business Need Identification — Recognition of a problem or opportunity
- Strategic Initiative — Alignment with organizational goals
- Stakeholder Request — Formal request from business units
- Technological Opportunity — Leveraging new technologies

The deliverables which signify software readiness for the next stage depend on the project management approach and may include:

- **Business requirements document** — BRD — a high level business goals specification which elicits a set of business functionalities that the software needs to meet in order to succeed
- **Software requirements specification** — SRS — the document to clearly define and record the software requirements and get them approved by the customer or the market analysts
- **Functional requirements document** — FRD — the specification which includes detailed system behaviors
- **User stories and Use cases** — the documents to describe features from an end-user perspective

Software requirements specification is commonly treated as the key document of the software Conception stage.

3.2. Software Planning Stage

Software Planning is the stage where the project's vision, scope, and feasibility are formally defined before any development begins.

This stage sets the trajectory for the entire project, ensuring alignment between stakeholders, realistic expectations, and a clear roadmap for execution.

Primary Goals

The primary goals of the Planning stage include:

- Defining project scope — what will and won’t be built
- Assessing feasibility — technical, economic, operational
- Estimating costs, timelines, and resources
- Identifying risks and mitigation strategies
- Establishing stakeholder alignment
- Creating a formal project plan

Key Activities

The Planning Stage consists of the following primary activities:

- **Feasibility Study** — a structured analysis to determine if the project is viable, which answers the following questions:
    - Technical Feasibility — Can it be built with chosen technology?
    - Economic Feasibility — How expensive will the product be?
    - Operational Feasibility — Will end-users adopt the solution?
    - Legal Feasibility — Does it comply with regulatory rules?
- **Scope Definition**:
    - In-Scope or Out-of-Scope — Clear definition of the project boundaries

    - Work Breakdown Structure — WBS — Hierarchical decomposition of deliverables
    - Scope Statement — Formal agreement to prevent "scope creep"

- **Risk Assessment and Mitigation**:
    - Risk Identification — Potential threats definition
    - Risk Analysis — Impact and likelihood evaluation
    - Mitigation Strategies — Contingency planning for high-priority risks

- **Resource Planning**:
    - Team Structure — Developers, testers, PMs and other roles with their responsibilities
    - Technology Stack — Programming languages, frameworks, databases
    - Budget Estimation — Development, testing, infrastructure, and maintenance costs

- **Timeline and Milestone Planning**:
    - Gantt Charts and Roadmaps — Visual timelines for each SDLC stage
    - Agile Sprints — Sprint durations and backlog priorities definition
    - Critical Path Method — CPM — Identification of the tasks that could delay the project

- **Selection of Development Methodology** — SDLC model choice based on the project needs

- **Approval and Kickoff**:
    - Project Charter — Authorizes the project and allocates resources
    - Stakeholder Sign-Off — Formal agreement on scope, budget, and timeline

    - Kickoff Meeting — Aligns all teams on objectives and processes

The outcome of the technical feasibility study is to define the various technical approaches that can be followed to create the app successfully.

The comprehension of quality assurance requirements and identification of the risks associated with the project are the crucial tasks of the planning stage either.

Entry and Exit Criteria

The previous stage deliverables serve as a natural trigger for the software Planning stage to begin.

The following deliverables of the Planning stage, in its turn, serve as a sign to exit the stage:

- **Project plan** — Provides overall roadmap with timelines, milestones, and resources
- **Feasibility report** — Justifies project viability: technical, financial, legal
- **Risk management plan** — Identifies risks and mitigation strategies
- **Scope statement** — Defines project boundaries and exclusions
- **Resource allocation plan** — Details team structure, tools, and budget
- **Communication plan** — Specifies how stakeholders will receive updates: meetings, reports, etc.

Planning Stage Role

The Planning Stage is the cornerstone of successful software development, transforming abstract ideas into actionable strategies.

By investing time in thorough planning, teams can avoid common pitfalls, optimize resources, and deliver projects on time and within budget.

3.3. Software Design Stage

Software Design is the stage to describe the desired features and operations of the system.

The aim of the Design stage is to figure out the type of clients and servers necessary for technical feasibility of the system.

Key Activities

The stage consists of the following primary activities:

- System architecture design
- IT infrastructure design

Software design may include:

- Hierarchy diagrams
- Screen layouts for user interfaces
- Entity-relationship diagrams — ERD
- Process diagrams
- Data dictionaries
- Business rules
- Pseudo code

— and other necessary insights gathered under the Software design specification.

Primary Goals

The Design stage has to describe a system as a collection of modules — or subsystems — according to requirements identified in the approved Software requirements specification.

Software design is to clearly define all architectural modules of the product along with its communication and data flow representation with the external and third party modules.

Exit Criteria

The completed Software Design Specification is the main exit criterion during the software Design stage.

Software design specification — SDS — is the representation of software design dedicated to store the design information, address various concerns, and to communicate the collected data to the design stakeholders.

The Software design specification often called:

- Software design description
- Software design document
- just a Design document

— includes, as a rule, an **Architecture Diagram** with reference to the smaller pieces of design.

3.4. Software Development Stage

Software Development is the stage where abstract designs and requirements are transformed into functional, executable code.

The primary objective of the Development stage is to translate detailed design specifications into a working software product through systematic programming, while ensuring code quality, maintainability, and alignment with requirements.

The Development stage includes following primary components to be evolved:

- Code
- Databases
- Infrastructure

Key Activities

The primary Development stage activities should usually include:

- **Coding or Programming** — Writing source code in chosen programming languages — Java, Python, C#, JavaScript, etc
- **Unit Testing** — Developers test individual components, called modules, to ensure they work correctly in isolation
- **Code Review** — Peer examination of code to improve quality, share knowledge, and detect defects early
- **Version Control** — Managing code changes using systems like Git, SVN, or Mercurial
- **Integration** — Combining individual software modules into a complete system

The supplementary Development stage activities may also include:

- **Database Implementation** — Creating and populating databases according to design specs
- **API Development** — Building internal and external interfaces
- **Configuration Management** — Managing environment-specific settings and parameters
- **Debugging** — Identifying and fixing code-level defects

Software and hardware are to be purchased and installed during this stage either.

Inputs and Outputs

The developers base their work on the following inputs:

- **Detailed Design Documents** — Technical specs, database schemas, API contracts
- **UI/UX Designs** — Wireframes, mockups, style guides
- **Architecture Diagrams** — System components and their interactions
- **Development Environment Setup** — Servers, IDEs, tools

The main deliverables of Development stage usually include:

- **Source Code** — Version-controlled, documented codebase
- **Unit Test Cases and Results** — Automated tests with pass/fail reports
- **Technical Documentation** — Code comments, API documentation
- **Build Artifacts** — Executable files, packages, containers

Programming languages

During the Development stage the programming language — PL — is chosen according to the type of the software developed.

To generate the code, developers must also follow the coding guidelines defined by their organization and such programming tools like compilers, interpreters, and debuggers.

Exit Criteria

The Development stage has its distinct definition of done:

- All features implemented according to specifications
- Unit tests written and passing with the target coverage met
- Code review completed for all changes
- Integration testing successful
- Technical documentation updated
- Ready for the upcoming Testing stage

3.5. Software Testing Stage

Software Testing is the stage dedicated to systematically evaluate and validate that a software meets specified requirements, functions correctly, and delivers a quality user experience.

Testing activities are mostly involved in all stages of the Software development life cycle.

Testing stage, however, refers to the product testing only.

The Software Testing stage serves as a primary **Quality Gate** before release to production as the defects are searched and managed until the product reaches the quality conditions defined in the Software requirements specification.

Key Activities

During the Testing stage, all pieces of code are integrated and deployed in the dedicated environment for the QA engineers to check the software for errors, flaws, and defects and to verify it functions as expected.

Testing stage includes the following primary activities:

- **Test Planning and Design**:
    - Test Strategy Development — Overall approach and objectives definition
    - Test Case Creation — Detailed test scenarios with steps and expected results design
    - Test Data Preparation — Test datasets creation and management
    - Test Environment Setup — Hardware, software, and networks configuration

- **Test Execution and Evaluation**:
    - Test Case Execution — Test runs according to created test plans
    - Defect Reporting — Identified issues being logged, tracked, and managed
    - Results Analysis — Outcomes against expected results evaluation
    - Regression Testing — Verification that new changes don't break existing functionality
- **Test Reporting and Closure**:
    - Test Summary Reports — Testing activities and outcomes documentation
    - Metrics Collection — Test coverage, defect density, etc data gathering
    - Exit Criteria Evaluation — Software is ready for release estimation

Testing Metrics and Measurements

Software testing metrics are quantitative measures used to evaluate the effectiveness, efficiency, progress, and quality of the testing process.

They provide data-driven insights to make informed decisions, identify improvement areas, and assess testing performance against objectives:

- Test Coverage — Percent of requirements covered by test cases
- Defect Density — Number of defects per size or metric
- Test Case Effectiveness — Percent of defects found by test cases
- Defect Leakage — Defects found post-release vs during testing

Entry and Exit Criteria

Entry Criteria define when Testing stage can begin:

- Requirements are stable and approved
- Development is substantially complete
- Test environment is ready
- Test cases are prepared and reviewed

Exit Criteria determine when Testing stage is to conclude:

- All critical and major defects are resolved
- Test coverage targets are met
- Performance and security benchmarks are achieved
- Stakeholder approval obtained

Testing Stage Role

Testing is the crucial part of software development life cycle which can save a lot of rework, time, and money.

To provide quality software, an organization must perform testing in a systematic way.

3.6. Software Deployment Stage

Software Deployment is the stage where the validated application is released and made operational in the production environment enabling end-users to access and use the system.

The Deployment stage, also called:

- Delivery stage
- Implementation stage
- Installation stage

— represents the transition from development to operational use in the appropriate market.

Key Activities

The Deployment stage commonly includes following key activities:

- **Pre-Deployment Preparation**:
    - Deployment Planning — Create detailed rollout strategy, schedule, and rollback plans
    - Environment Setup — Configure servers, databases, networks, and security settings
    - Final Verification — Conduct smoke tests and sanity checks in production-like staging
    - Backup Creation — Backup existing systems and data before deployment
    - Stakeholder Communication — Notify users, support teams, and stakeholders about upcoming changes

- **Deployment Execution**:
    - Package Deployment — Install application binaries, libraries, and dependencies
    - Database Migration — Execute SQL scripts, data transfers, and schema updates
    - Configuration Application — Set environment-specific parameters and settings
    - Integration Activation — Enable connections to external systems and Application program interfaces, APIs
    - Service Initialization — Start application services and background processes
- **Post-Deployment Validation**:
    - Health Checks — Verify all services are running correctly
    - Functional Testing — Confirm critical business functions work in production
    - Performance Validation — Ensure response times meet Service Level Agreement, SLA, requirements
    - User Access Testing — Verify authentication and authorization mechanisms
    - Monitoring Setup — Configure alerts, logs, and performance monitoring

Deployment Strategies

The Deployment strategies include:

- **Traditional approaches**:
    - Big-Bang Deployment — Complete system replacement at once

    - Parallel Deployment — Old and new systems run simultaneously
    - Incremental Deployment — Roll out by modules, regions, or user groups

- **Modern — Continuous Deployment — approaches**:
    - Blue-Green Deployment — Two identical environments to switch traffic between them
    - Canary Release — Gradually expose new version to small user subset
    - Feature Flags — Deploy code with features toggled off to enable them gradually
    - Rolling Deployment — Incrementally update instances while maintaining service

Entry and Exit Criteria

The prerequisites for Deployment to begin are as follows:

- All testing stages completed successfully
- Stakeholder approval obtained — business sign-off
- Production environment prepared and validated
- Rollback plan documented and tested
- User documentation and training completed

The Deployment completion prerequisites are as follows:

- Application successfully running in production
- All integration points functioning correctly
- Performance benchmarks met
- Monitoring and alerting operational
- Support teams trained and ready
- Post-deployment review conducted

3.7. Software Maintenance Stage

Software Maintenance is the stage for adjustments, amendments, and enhancements designated to keep software updated, operable, and performant after its initial deployment.

Key Objectives

The primary goal of the Maintenance Stage is to recurrently update and upgrade software to adapt it for the future challenges.

As a whole, Maintenance is mostly aimed to:

- Correct Issues — Identify and fix bugs discovered in production
- Adapt to Changes — Modify software to work in changing environments
- Enhance Functionality — Add new features and improve existing capabilities
- Prevent Degradation — Optimize performance and address technical debt
- Ensure Continuity — Maintain compatibility with evolving platforms and standards

Key Activities

The Maintenance stage activities may be divided into the following categories:

- **Corrective Maintenance**:
    - Defect Resolution — Fix bugs and errors reported by users
    - Emergency Fixes — Address critical issues affecting system availability
    - Patch Management — Deploy small, focused updates

- **Adaptive Maintenance**:
    - Platform Updates — Adapt to new operating systems, hardware, or cloud platforms
    - Third-Party Integration — Maintain compatibility with external systems
    - Regulatory Compliance — Address legal and regulatory requirements
- **Perfective Maintenance**:
    - Performance Optimization — Improve speed, efficiency, and resource usage
    - Usability Enhancements — Improve user interface and experience
    - Feature Additions — Implement new functionality based on user feedback
- **Preventive Maintenance**:
    - Code Refactoring — Restructure code without changing functionality
    - Technical Debt Reduction — Address shortcuts taken during development
    - Documentation Updates — Keep documentation current with system changes

Maintenance Models

Maintenance models may be divided into the following groups:

- **Traditional Models**:
    - Quick-Fix Model — Immediate repairs without detailed analysis

    - Iterative Enhancement — Cyclical improvement through analysis, redesign
    - Reuse-Oriented — Leverage existing components for maintenance
- **Modern Approaches**:
    - Agile Maintenance — Regular maintenance sprints with user feedback
    - DevOps Maintenance — Integrated development and operations teams
    - AI-Driven Maintenance — Predictive analytics for issue prevention

Maintenance Stage Role

Software Maintenance is not merely about fixing bugs but encompasses a strategic, ongoing process of keeping software valuable, secure, and aligned with business needs.

The most successful organizations treat maintenance as a strategic investment rather than a necessary cost.

Effective maintenance requires balancing reactive support with proactive improvement, ensuring that software continues to deliver value throughout its lifecycle.

Maintenance is not the final stage but is a soft return to the Conception stage with the intent of further enhancements on the next level of the Software development life cycle.

3.8. SDLC Models

Software Development Life Cycle models are shortly referred to as Software Development Process Models.

Software Development Process Model — SDPM — is a structured framework that defines the sequence of activities, tasks, and workflows for developing software products.

SDPM provides a systematic approach to transforming user requirements into functional software while managing constraints like time, cost, and quality.

Each SDPM follows a series of steps unique to its type in order to ensure the success of the software development flow.

SDPM Approaches

All SDPMs may be classified into two big categories according to the development approaches used:

- Predictive
- Adaptive

Predictive SDPMs

Predictive SDPMs — or **Plan-Driven SDPMs** — are the models which assume that requirements can be fully defined at the beginning of the project and will remain relatively stable during development.

They focus on analyzing and planning the future in detail and cater for known risks. In the extremes, a predictive team can report exactly what features and tasks are planned for the entire length of the development process.

Predictive models rely on effective early-stage analysis and if the stage fails, the project may face difficulties while changing direction.

Adaptive SDPMs

Adaptive SDPMs — or **Value-Driven SDPMs** — are the models which recognize that requirements evolve during development and emphasize flexibility, collaboration, and continuous improvement.

They focus on adapting quickly to changing realities — when the needs of a project change, an adaptive team changes as well.

An adaptive team has difficulty describing exactly what will happen in the future — the further away a date is, the more vague an adaptive model is about what will happen on that date.

IV. Predictive SDPMs

4. Predictive SDPMs

As the name suggests, Predictive SDPM assumes one can predict the complete workflow.

It involves fully understanding the final product and determining the process for delivering it.

In this form of project life cycle, one determines the cost, scope, and timeline in the early stages of the project.

Benefits

The main benefits of Predictive SDPMs are as follows:

- It is easy to understand and follow as each stage is initiated after the previous one is completed
- The laid down instructions and concise workflow makes it easier for the developers to work within a specified budget and timeframe
- If everything goes as planned it enables organizations to assume the expected project budget and timelines
- Each stage has specific timelines and deliverables which makes it easier for teams to operate and monitor the entire project
- The main concern of a predictive approach is to develop and maintain the specifications of the final product

This makes it ideal for projects where all the requirements are defined and well understood with a clear vision of the final product.

Within the Predictive approach, there are minimal expected changes as the work is already predicted and well-known.

The team has a clear idea of exactly where the project is heading and how to follow the sequence.

Drawbacks

The main drawbacks of the Predictive approach are as follows:

- Working software is produced at a later stage, which leads to delayed identification of bugs and vulnerabilities in the application
- Organizations often have to bear additional costs of delayed applications if bugs are discovered in the testing stage of the project
- Complex projects are poorly manageable: it is not suitable for dynamic projects that entail flexible requirements or uncertainty in the end product

To sum up, a predictive approach can be extremely rigid, requiring developers to maintain strict and rigorous standards throughout the life cycle.

Since the sequence of the work is already predetermined, any subsequent changes can be very costly and time-consuming.

Predictive models

Below are the most important predictive Software Development Process Models:

- Big Bang Model
- Waterfall Model
- Incremental Model
- Iterative Model
- Spiral Model
- V-Model

4.1. Bing Bang Model

Big Bang Model is a process of software development focusing on all types of resources in software development and coding, with no or very little planning.

The requirements are understood and implemented when they come.

Key Characteristics

The Big Bang Model is an SDPM following no specific process or procedure and there is very little planning required.

The development starts with the money and efforts required as the input, and the software developed — as the output.

Even the customer may not be sure about his requirements and they are implemented on the fly without much analysis.

Usually the Big Bang Model is followed for small projects where the development teams are very small.

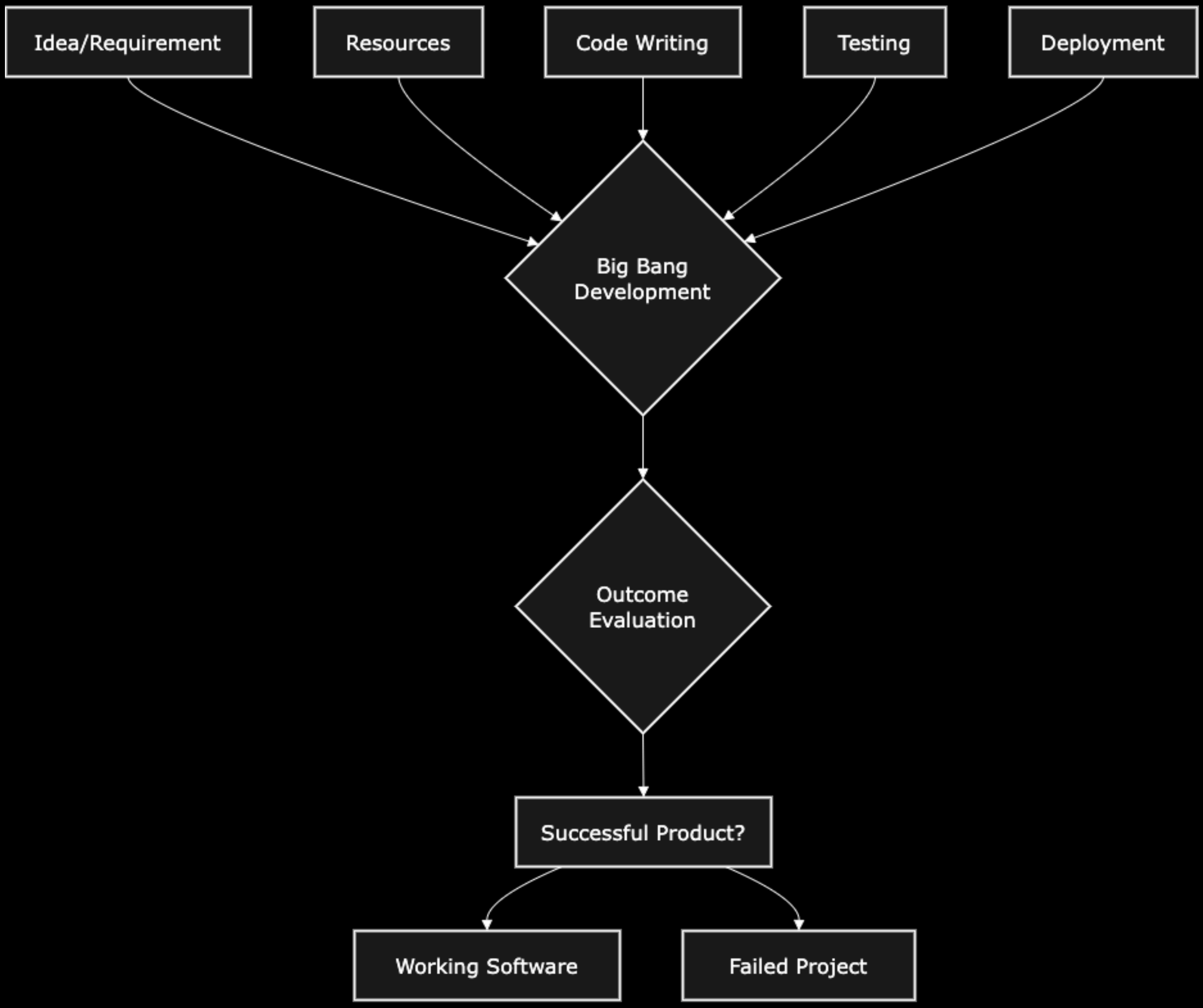
Key Activities

The Big Bang Model focuses all the possible resources on the software development, with very little or no planning.

The requirements are understood and implemented as they appear; any changes required may or may not need to revamp the complete software.

This model is ideal for small projects with one or two developers working together and is also useful for academic or practice projects.

It is also an ideal model for the product where requirements are not well understood and the final release date is not given.



Bing Bang Model

Benefits

The Big Bang Model is:

- Simple — Requires very little or no planning
- Manageable — Requires no formal procedure
- Affordable — requires very few resources
- Flexible — Requires lesser qualification

To conclude, the Big Bang model is ideal for repetitive or small projects with minimum risks.

Drawbacks

The main Drawbacks of the Big Bang Model are as follows:

- There are very high risk and uncertainty
- It excludes complex and object-oriented projects
- It rejects long and ongoing projects
- It may become very expensive if requirements are misunderstood

So, the Big Bang Model is of a very high risk as misunderstood or changed requirements may lead to complete reversal or scrapping of the project.

4.2. Waterfall Model

Waterfall Model is a process of software development that divides the whole software development life cycle into various stages.

The Waterfall SDPM was the first model to be introduced in 1970 by Winston Royce and is also referred to as a Linear-sequential SDPM.

Key Activities

Waterfall model is a sequential SDPM that divides software development into pre-defined stages.

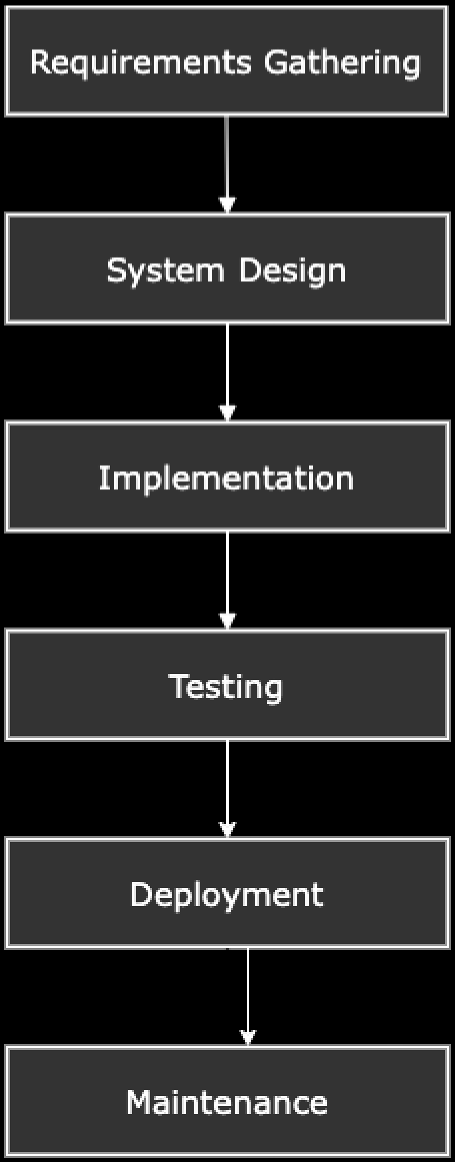
Each stage is designed for a specific activity and must be completed before the next stage can begin with no overlap between the stages.

Applicability

Every software is different and requires a suitable SDPM to be followed based on the internal and external factors.

Some situations where the use of Waterfall Model is most appropriate are:

- Requirements are thoroughly documented, clear, and fixed
- Technologies and tools involved are familiar and consistent
- Experienced resources are available and ample
- Application is simple and compact
- Software environment is stable
- Project term is concise



Waterfall Model

Benefits

The Waterfall model is the earliest SDPM that was used for software development, it is very simple to understand and use.

Some of the major advantages of the Waterfall Model are as follows:

- Clearly defined stages
- Well understood milestones

- Easily to arrange tasks
- Well documented processes
- Ready to manage resources
- Deliverables at every stage
- One at a time stage run
- Handily reviewed results

Drawbacks

The major disadvantages of the Waterfall Model are as follows:

- Intensive untested documentation
- Operating software appears late in SDLC, hence, testing stage starts too late
- Project changes are unpredictable
- Risk and uncertainty are of high levels
- Complex or object-oriented projects are excluded
- Long and ongoing projects are incapable
- Projects of changing requirements are poorly managed
- Project run requires environment stability support
- Integration is complicated and hassled
- Stage progress is difficult to measure
- Technological bottlenecks are poorly identifiable
- Business challenges are difficult to overwhelm

4.3. Incremental Model

Incremental Model is a process of software development where requirements are broken down into multiple standalone modules of software development life cycle.

Key Activities

Incremental SDPM may be treated as a series of waterfall cycles.

The requirements are divided into groups at the start of the project; for each group, the Incremental model is followed to develop software.

Process

Each release adds more functionality until all requirements are met; every cycle serves the maintenance stage for the previous release.

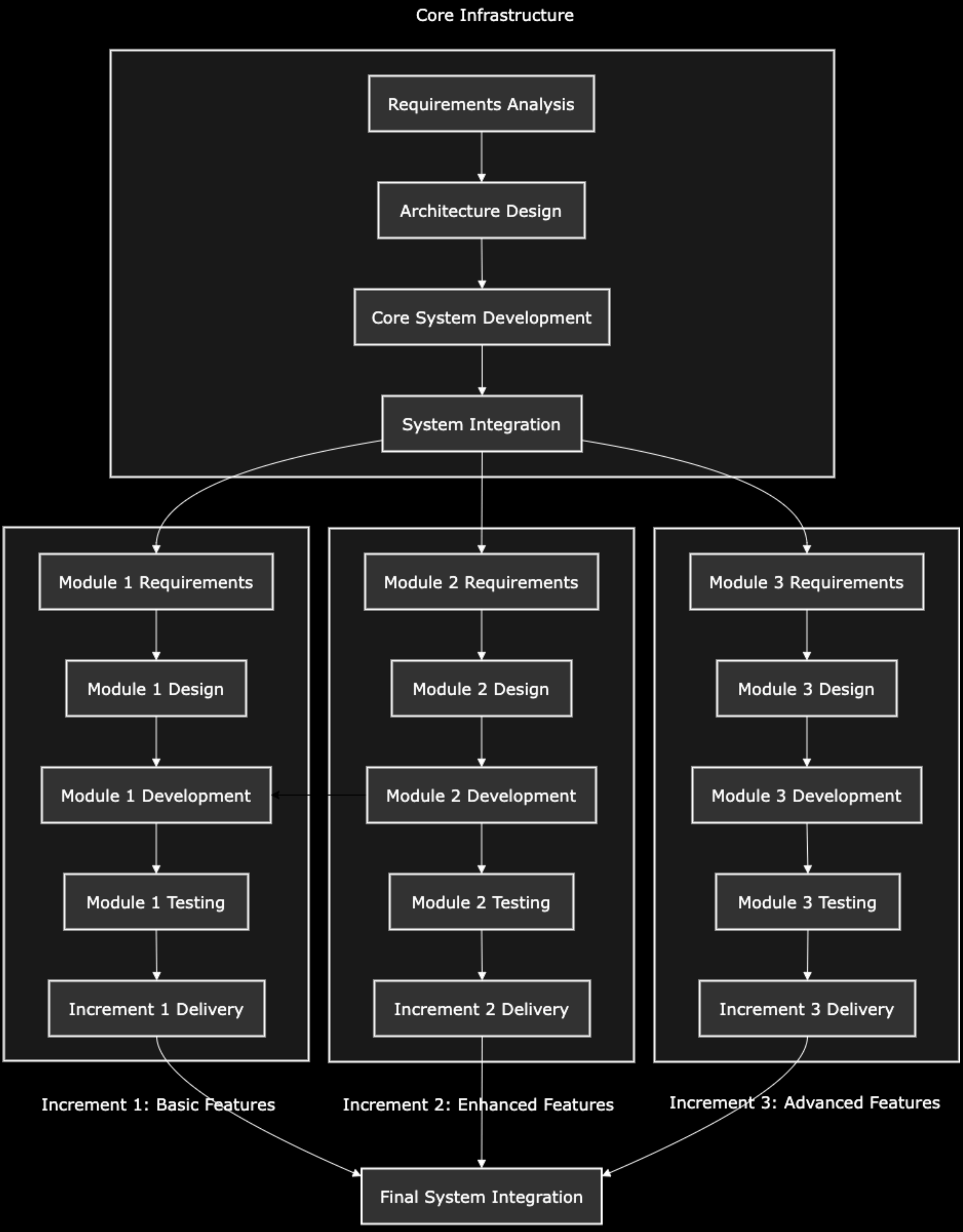
Incremental Model modifications allow the development cycles to overlap, so that a subsequent cycle may begin before the previous one is complete.

Iterations

Each iteration passes through the next four stages:

- Requirements Analysis
- Design
- Coding
- Testing

and each subsequent release of the system adds function to the previous release until all designed functionality has been implemented.



Incremental Model

Procedure

The software is put into production when the first increment is delivered.

The first increment is often a core product where the basic requirements are addressed, and supplementary features are added in the next increments.

Once the core product is analyzed by the client, there is a development for the next increment.

Model Characteristics

The main Incremental Model characteristics include:

- Software development is broken down into many mini development projects
- Partial systems are successively built to produce a final total system
- Highest priority requirement is tackled first
- Once the requirement is developed, requirement for that increment are frozen

Applicability

Incremental Model may be used when:

- Software requirements are clearly understood
- Early release of a product is of significant value
- There are features and goals of considerable risk
- Developers are not highly skilled or trained
- Software is developed by product company
- Application is web based and may seamlessly be updated

Benefits

The main advantages of the Incremental Model are as follows:

- High speed of development
- High flexibility
- Lower expenses
- Changes are possible
- Customer can respond to each building
- Errors are easy to be identified

Drawbacks

Below are the main pitfalls of the Incremental Model:

- farsighted planning is required
- each iteration cycle is rigid enough
- any unit bugfix influences all the software
- bugfixes are numerous and consume a lot of time

4.4. Iterative Model

Iterative Model — or **Evolutionary Acquisition Model** — is the process of software development which starts with an implementation of a small simple subset of requirements and iteratively enhances the evolving versions until the complete application is implemented and ready to be deployed.

Key Activities

The Iterative SDPM does not attempt to start with a full specification of requirements.

Instead, development begins by specifying and implementing just part of the software, which is then reviewed to identify further requirements.

This process is then repeated, producing a new version of the software at the end of each iteration of the model.

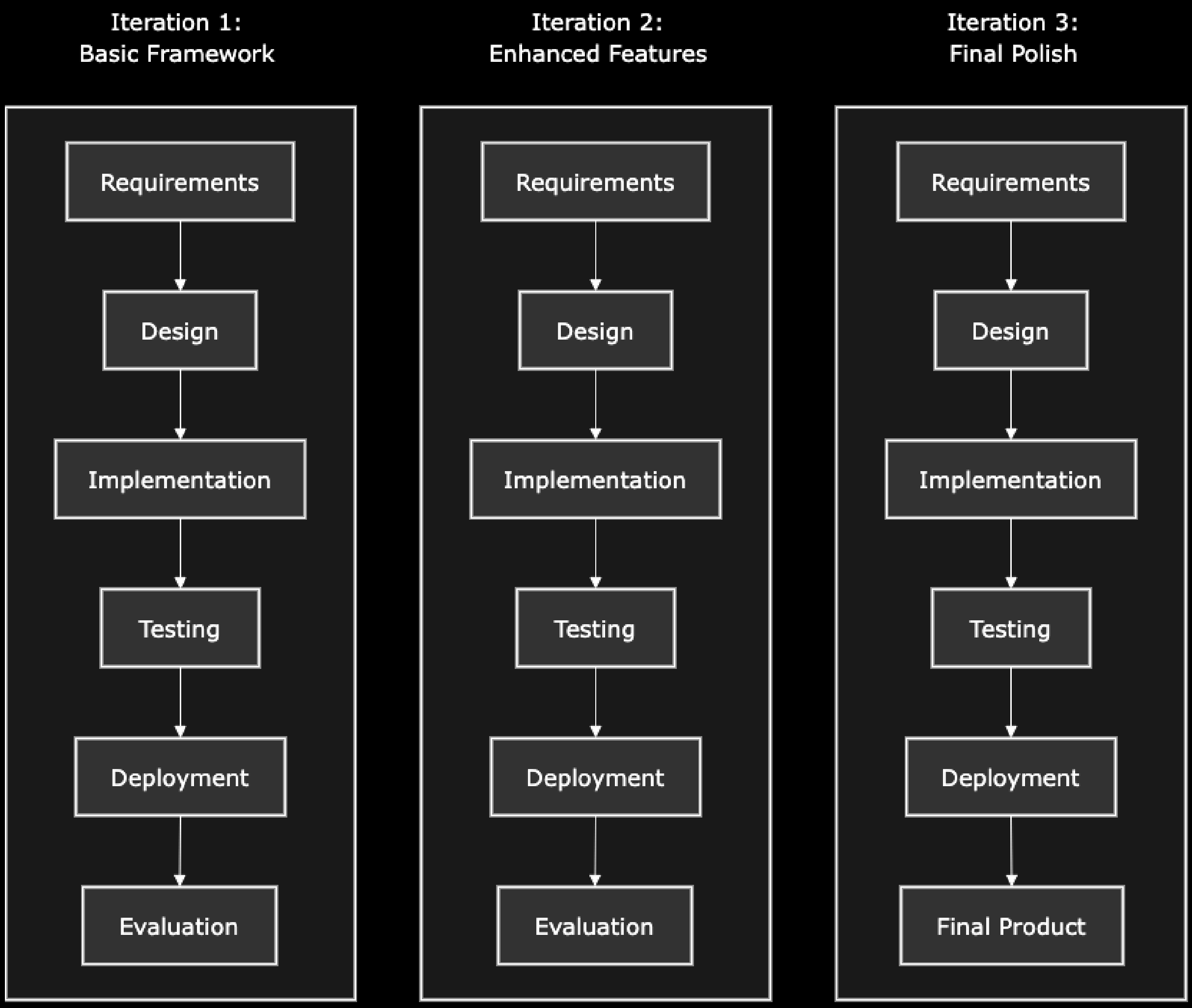
At each iteration, design modifications are made and new functional capabilities are added.

The basic idea behind this method is to develop a system through repeated, iterative, cycles and in smaller, incremental, portions at a time.

Workflow

Iterative Model is a combination of iterative design and incremental development.

During software development, more than one iteration of the software development cycle may be in progress at the same time.



Iterative Model

Builds

Iterative Model implies all the requirements are divided into various builds.

During each iteration, the module developed goes through the requirements analysis, design, implementation, and testing stages.

Each subsequent release of the module adds a new functionality to the previous one.

The process continues until the complete system is ready as per the requirement.

Testing Role

The key features of the Iterative Model are rigorous validation of requirements, and both verification and testing of each version of the software against those requirements within each iteration.

As the software evolves through successive cycles, tests must be repeated and extended to verify each version of the software.

Applicability

Like other SDPMs, Iterative development has some specific applications in the software industry.

This model is most often used in the following scenarios:

- Product requirements are clearly defined and understood
- New technologies are to be adopted by development team
- Certain functionalities or requested enhancements may evolve during project
- Features and goals of a high risk may appear in the future
- Specific iterations' resources are to be outsourced

Benefits

The main advantage of the Iterative Model is the operating software gained at the earliest stages of development.

The operating software, in particular, makes it easier to find functional or design flaws.

The flaws found early in the life cycle enable developers to save budget while software adjustments are applied.

All the advantages of the Iterative Model are as follows:

- Large and mission-critical projects are performable
- Operating software appears early in the life cycle
- Customer enjoyment, evaluation, and feedback start shortly
- Every increment delivers a new software functionality
- The results obtained are immediate and periodical
- Each iteration serves a manageable milestone
- Software development progress is measurable
- Parallel development is enabled
- Smaller iterations simplify testing and debugging
- Requirement changes are easily governed at lesser expences
- Issues, challenges, and risks found may be treated incrementally
- Iterated risk analysis, identification, and resolution is steadier
- High risks may be managed first

Drawbacks

The main drawback of the Iterative Model is it only fits large software development projects: breaking a small system into further small serviceable increments or modules is hard.

Other disadvantages of the Iterative Model are as follows:

- Model may require more resources to apply
- Often requirement changes keep the adoption cost high
- Serious system architecture or design issues may arise
- Management attention is of a greater need and complexity
- Risk analysis requires highly skilled resources
- Progress is highly dependent on the risk analysis stage
- Increment definitions redefine the complete system
- Risky level of uncertainty at the end of the project
- Model is poorly suitable for smaller projects

4.5. Spiral Model

Spiral Model is the combination of Iterative Model with the systematic, controlled aspects of the sequential linear development inherent to the Waterfall Model.

In other words, the model is a conjunction of Iterative and Waterfall Models with a distinct emphasis on risk analysis.

It allows incremental releases of the product — or **Incremental Refinement** — through each iteration around the spiral.

Spirals

The Spiral SDPM includes the four stages described below.

A software project repeatedly passes through these stages in iterations called Spirals.

Based on the customer evaluation, the software development process enters the next iteration and subsequently follows the linear approach to implement the feedback suggested by the customer.

The process of iterations along the spiral continues throughout the life of the software.

1. Identification

This stage starts with gathering the business requirements in the baseline spiral.

In the subsequent spirals, as the product matures, identification of system, subsystem, and unit requirements are all done in this stage.

This stage also includes understanding the system requirements by continuous communication between the customer and the system analyst.

At the end of the spiral, the product is deployed in the identified market.

2. Design

The Design stage starts with the conceptual design in the baseline spiral and involves architectural design, logical modules design, physical product design and the final design in the subsequent spirals.

3. Construct or Build

The Construct stage refers to production of the actual software product at every spiral.

In the baseline spiral, when the product is just thought of and the design is being developed a Proof of Concept — POC — is developed in this stage to get customer feedback.

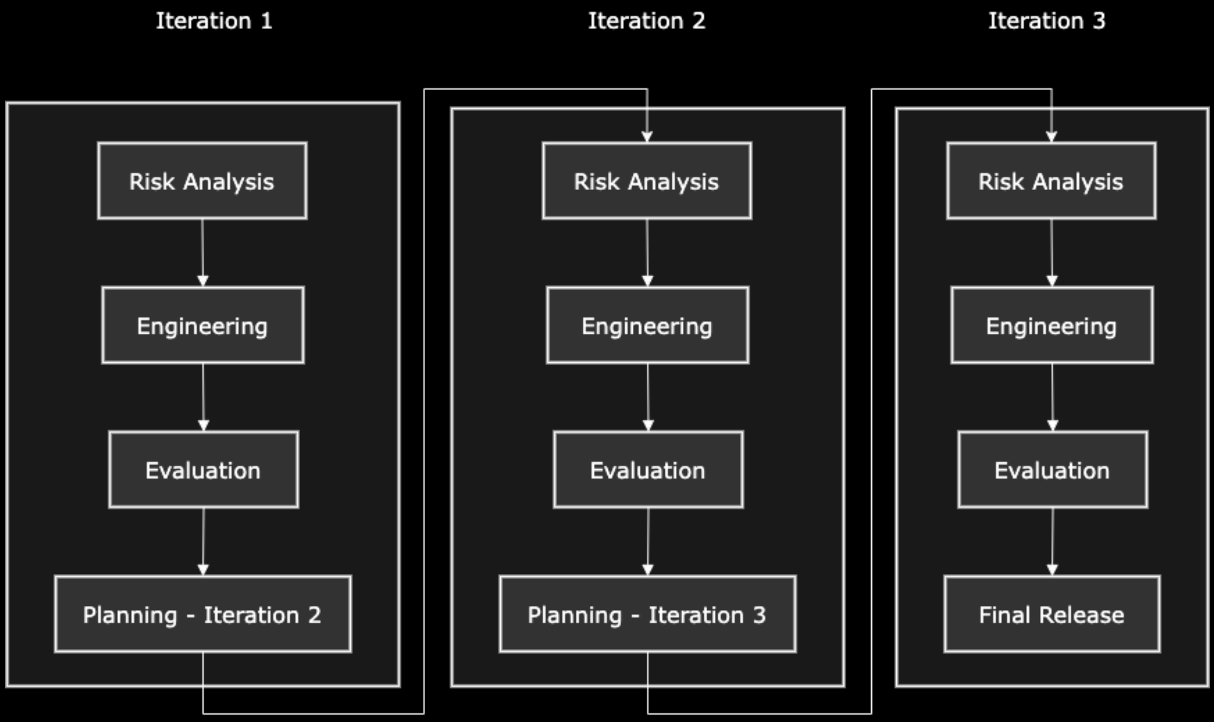
In the subsequent spirals with higher clarity on requirements and design details a working software model, called Build, is produced with a version number.

These Builds are sent to the customer for feedback.

4. Evaluation and Risk Analysis

Risk Analysis includes identifying, estimating and monitoring the technical feasibility and management risks, such as schedule slippage and cost overrun.

After testing the Build, at the end of the first iteration, the customer evaluates the software and provides feedback.



Spiral Model

Key Activities

Each stage of the Spiral Model in software engineering begins with a design goal and ends with the client reviewing the progress.

The development process starts with a small set of requirements and goes through each of the Waterfall Model stages.

The software engineering team adds functionality for the additional requirement in every-increasing spirals until the software is ready for the production stage.

Benefits

The advantage of the Spiral Model is that it allows elements of the product to be added in when they become available or known.

This assures that there is no conflict with previous requirements and design.

This method is consistent with approaches that have multiple software builds and releases which allows making an orderly transition to a maintenance activity.

Another positive aspect of this method is that the Spiral model forces an early user involvement in the system development effort.

Else advantages of the Spiral model are as follows:

- Model allows extensive use of prototypes
- Fragmented prototype building simplifies the cost estimation
- Continuous repeated development streamlines the risks management
- Faster development and more regular improvements
- Customers gain the system early for use and feedback
- Requirement changes are easily accommodated
- Requirements are gathered more accurately
- Additional functionality may be shifted for a later stage
- Development may be divided into smaller parts
- Risky parts of software may be developed earlier

Drawbacks

The main disadvantage of the Spiral Model is that it takes a very strict management to complete such apps and there is a risk of running the Spiral in an indefinite loop.

The discipline and the extent of taking change requests is very important to develop and deploy the product successfully.

Other disadvantages of the Spiral Model are as follows:

- Schedule or budget discrepancies are of a high probability
- A lot of intermediate stages requires excessive documentation
- Model protocol should be followed strictly to smooth operation
- The process and its management are more complex
- Risk assessment expertise is obvious for the model
- Spiral development suits best the large projects only

- Mismatches the smaller projects due to excessive costs
- Low risk projects misfit the model due to unnecessary complexity
- Spiral may continue indefinitely

Applicability

The Spiral Model is widely used in the software industry as it is in sync with the natural development process of any product — learning with maturity which involves minimum risk for the customer as well as the development firms.

The following pointers explain the typical uses of a Spiral Model:

- Project is large
- Releases are to be frequent
- Customer feedback is necessary
- Prototype creation is possible
- Risk evaluation is important
- Budget constraints are weighty
- Business priority changes are often
- Project is of a medium or high risk
- Requirements are foggy, variable, or complex
- Project revision may happen any time
- Significant changes are expected

4.6. V-Model

V-Model is an extension of the Waterfall SDPM and is based on the association of a testing stage for each corresponding development stage.

The model consists of Verification and Validation stages, and is also known as **Verification and Validation Model**.

Key Activities

V-Model is the SDPM where process execution happens in a V shape.

It means that every development activity has its directly associated testing stage done in parallel in a sequential way.

This is a highly disciplined model where the next stage starts only after completion of the previous one.

Verification Stages

There are several Verification stages in the V-Model, each of these are explained in detail below.

1. Business Requirements Analysis

This is the first stage in the development cycle where the product requirements are understood from the customer's perspective.

This stage involves detailed communication with the customer to understand his expectations and exact requirements.

This is a very important activity and needs to be managed well, as most of the customers are not sure about what exactly they need.

The acceptance test design planning is done at this stage as business requirements can be used as an input for acceptance testing.

2. System Design

Once the product requirements are clear and detailed, it is time to design the complete system.

The system design will have the understanding and detailing the complete hardware and communication setup for the product under development.

The system test plan is developed based on the system design.

Doing this at an earlier stage leaves more time for the actual test execution later.

3. Architectural Design

Architectural specifications are understood and designed in this stage.

Usually more than one technical approach is proposed and based on the technical and financial feasibility the final decision is taken.

The system design is broken down further into modules taking up different functionality.

This is also referred to as High Level Design — HLD.

The data transfer and communication between the internal modules and with the outside world — other systems — is clearly understood and defined in this stage.

With this information, integration tests can be designed and documented during this stage.

4. Module Design

At this stage, the detailed internal design for all the system modules is specified, referred to as Low Level Design — LLD.

It is important that the design is compatible with the other modules in the system architecture and the other external systems.

The unit tests are an essential part of any development process and help eliminate the maximum faults and errors at a very early stage.

These unit tests can be designed at this stage based on the internal module designs.

5. Coding Stage

The actual coding of the software modules designed during the Design stage is taken up in the Coding stage.

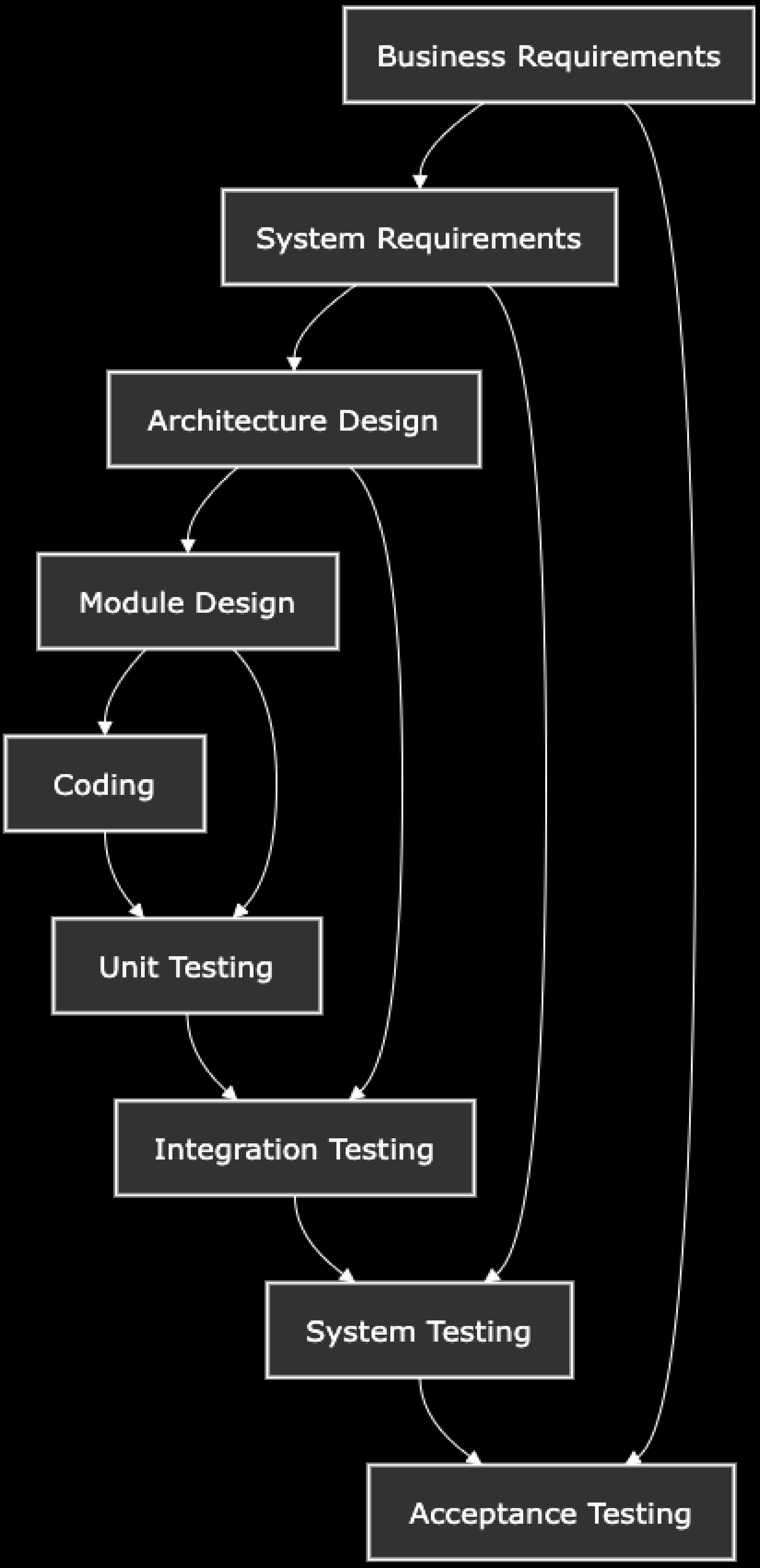
The best suitable programming language is decided based on the system and architectural requirements.

The coding is performed based on the coding guidelines and standards.

The code goes through numerous code reviews and is optimized for best performance before the final build is checked into the repository.

Validation Stages

The different Validation stages in a V-Model are explained in detail below.


Business Requirements
System Requirements
Architecture Design
Module Design
Coding
Unit Testing
Integration Testing
System Testing
Acceptance Testing

V-Model

1. Unit Testing

Unit tests designed in the module Design stage are executed on the code during this validation stage.

Unit testing is the testing at code level and helps eliminate bugs at an early stage, though all defects cannot be uncovered by unit testing.

2. Integration Testing

Integration testing is associated with the Architectural design stage.

Integration tests are performed to test the coexistence and communication of the internal modules within the system.

3. System Testing

System testing is directly associated with the System design stage.

System tests check the entire system functionality and the communication of the system under development with external systems.

Most of the software and hardware compatibility issues can be uncovered during this system test execution.

4. Acceptance Testing

Acceptance testing is associated with the Business requirements analysis stage and involves testing the product in a user environment.

Acceptance tests uncover the compatibility issues with the other systems available in the user environment.

It also discovers the non-functional issues such as load and performance defects in the actual user environment.

Benefits

The main advantage of the V-Model is that it is very easy to understand and apply.

The simplicity of this model also makes it easier to manage.

All advantages of the V-Model are as follows:

- Simple and easy to understand and apply
- Easy to manage due to the rigidity of the model
- Each stage has specific deliverables and a review process
- Allows smaller projects where requirements are well understood
- Stages are completed one at a time

Drawbacks

V-Model is not flexible to changes, which are common in a dynamic world, and the disadvantage is the greatest one.

The case requirement changes happen, it is very expensive to implement them.

All disadvantages of the V-Model are as follows:

- risk and uncertainty are of a high level
- complex and object-oriented projects are prevented
- long and ongoing projects are excluded
- projects where requirements are at a middle or high risk of changing are rejected
- at testing stage, the application is difficult to go back and change a functionality
- working software is absent until the closure of the life cycle

Applicability

V-Model applicability greatly resembles the one of the Waterfall model, as both models are of the sequential type.

Requirements have to be very clear before the project starts, because it is usually expensive to go back and make changes.

This model is often used in the medical development field, a strictly disciplined domain.

The following pointers are some of the most suitable scenarios to use the V-Model application:

- requirements are well defined, clearly documented and fixed
- product definition is stable
- technology is not dynamic and is well understood by the project team
- there are no ambiguous or undefined requirements
- the project is short in time

V. Adaptive SDPMs

5. Adaptive SDPMs

Adaptive SDPMs are the models which recognize that requirements evolve during development and emphasize flexibility, collaboration, and continuous improvement.

Adaptive SDPM have a mix of incremental and iterative development.

It involves adding features incrementally and making changes and refinements according to the feedback.

In other words, the work can easily adapt to the changing requirements based on new feedback received from the client.

Key Element

A key element of an Adaptive SDPM is that while it defines certain milestones throughout the SDLC, it also allows flexibility to achieve them.

Adaptive SDPM focuses on achieving the desired end goal by quickly adapting the dynamic business requirements.

It puts more focus on the present requirement and leaves room for future scope of the project.

Agility

All adaptive SDPMs are collectively referred to as Agile methods, after the **Agile Manifesto** was published in 2001.

The Agile thought had started early in the software development and got popular due to its flexibility and adaptability.

Agile Methods

Following are the most popular Agile Methods:

- Rapid application development — RAD
- Rapid prototyping
- Dynamic systems development
- Rational unified process
- Scrum
- Crystal Clear
- Extreme programming
- Feature driven development

Scrum is by far the most popular and de facto standard Agile development method, most likely because it's easy to implement and maintain.

5.1. Agile Methodology

Agile methodology is a combination of iterative and incremental process models with focus on process adaptability and customer satisfaction by rapid delivery of working software products.

Activities

Agile methods break the product into small incremental builds.

These builds are provided in iterations.

Each iteration typically lasts from about one to three weeks.

Every iteration involves cross functional teams working simultaneously on various areas like:

- Planning
- Requirements analysis
- Design
- Coding
- Unit testing
- Acceptance testing

At the end of the iteration, a working product is displayed to the customer and important stakeholders.

Time Boxes

Agile methodology believes that every project needs to be handled differently and the existing methods need to be tailored to best suit the project requirements.

In Agile, the tasks are divided into **Time Boxes** — small frames of time — to deliver specific features for a release.

Iterative approach is taken and working software build is delivered after each iteration.

Each build is incremental in terms of features.

The final build holds all the features required by the customer.

5.2. Agile Methods

Agile methods are being widely accepted in the software world. However, these methods may not always be suitable for all products.

Below are some benefits and drawbacks of the Agile methodology.

Benefits

The main advantages of the Agile methodology are as follows:

- Convenient teamwork and cross training
- Fast working solutions development and demonstration
- Minimal resource requirements
- Ready for fixed or changing requirements
- Steadily changed environments adoption
- Concurrent development and delivery enabled
- Minimum of planning, rules, and documentation
- Real flexibility for developers and simple management

Drawbacks

The main disadvantages of the Agile methodology are as follows:

- Poor handling for complex dependencies
- Sustainability, maintainability, and extensibility are at high risks
- Overall plan, leader, and Project manager are obvious
- Strict delivery deadlines for adjustments and added functionality
- Heavy dependency on customer interaction
- Challenging technology transfer and high individual dependency due to the documentation deficiency

5.3. Agile Manifesto

Agile Touchstones

Following are the touchstones of Agile Manifesto:

- Individuals and interactions over Processes and tools
- Working software over Comprehensive documentation
- Customer collaboration over Contract negotiation
- Responding to change over Following a plan

Agile Principles

Following are the principles behind the Agile Manifesto:

- Our highest priority is to satisfy the customer through early and continuous delivery of valuable software
- Welcome changing requirements, even late in development
- Agile processes harness change for the customer's competitive advantage
- Deliver working software frequently, from a couple of weeks to a couple of months, with a preference to the shorter timescale
- Business people and developers must work together daily throughout the project
- Build projects around motivated individuals. Give them the environment and support they need, and trust them to get the job done
- The most efficient and effective method of conveying information to and within a development team is face-to-face conversation
- Working software is the primary measure of progress
- Agile processes promote sustainable development. The sponsors developers, and users should be able to maintain a constant pace indefinitely

- Continuous attention to technical excellence and good design enhances agility
- Simplicity — the art of maximizing the amount of work not done — is essential
- The best architectures, requirements, and designs emerge from self-organizing teams
- At regular intervals, the team reflects on how to become more effective, then tunes and adjusts its behavior accordingly.

5.4. Rapid Application Development

Rapid application development — RAD — is the software development method based on prototyping and iterative development with no specific planning involved.

Key Activities

Rapid Application Development focuses on:

- Gathering customer requirements through workshops or focus groups
- Early testing of the prototypes by the customer using iterative concept
- Reuse of the existing prototypes — components
- Continuous integration and rapid delivery
- Routine operations automation
- Visual prototype programming

Key Objectives

Rapid application development is a software development methodology that uses minimal planning in favor of rapid prototyping.

A **Prototype** is a working model that is functionally equivalent to a component of the product.

In the RAD model, the functional modules are developed in parallel as prototypes and are integrated to make the complete product for faster product delivery.

Since there is no detailed preplanning, it makes it easier to incorporate the changes within the development process.

Workflow

RAD projects follow an iterative and incremental model and have small teams of developers, domain experts, customer representatives and other IT resources working progressively on their component or prototype.

The most important aspect for this model to be successful is to make sure that the prototypes developed are reusable.

Benefits

The main advantages of the RAD Model are as follows:

- Requirement changes may be accommodated
- Progress can be measured
- Iteration time can be short with use of powerful rad tools
- Productivity with fewer people in a short time
- reduced development time
- Increases reusability of components
- Quick initial reviews occur encouraging customer feedback
- Integration from very beginning solves a lot of integration issues

Drawbacks

The main disadvantages of the RAD Model are as follows:

- Dependency on technically strong team members for identifying business requirements
- Only system that can be modularized can be built using RAD
- requires highly skilled developers and designers
- High dependency on modelling skills
- Inapplicable to cheaper projects as cost of modelling and automated code generation is very high
- Management complexity is more suitable for systems that are component based and scalable

- Requires user involvement throughout the life cycle
- Suitable for projects requiring shorter development times

Applicability

The RAD model can be applied successfully to the projects in which clear modularization is possible.

If the project cannot be broken into modules, RAD may fail.

The following pointers describe the typical scenarios where RAD can be used:

- System can be modularized to be delivered incrementally
- There is a high availability of designers for modelling
- Budget permits use of automated code generating tools
- Domain experts are available with relevant business knowledge
- Requirements changes and working prototypes are to be presented to customer in small iterations of 2-3 months

RAD model enables rapid delivery as it reduces the overall development time due to the reusability of the components and parallel development.

RAD works well only if highly skilled engineers are available and the customer is also committed to achieve the targeted prototype in the given time frame.

If there is commitment lacking on either side the model may fail.

5.5. Rapid Prototyping

Rapid Prototyping refers to the building of software prototypes which display the functionality of the developed application.

The prototype, though, may not actually hold the exact logic of the original software.

Rapid Prototyping becomes very popular, as it enables the team to understand customer requirements at an early stage of development.

It also helps software designers and developers to get customer feedback and understand what exactly is expected from the product built.

Key Activities

Prototype is a working model of software with some limited functionality.

The Prototype does not always hold the exact logic used in the actual software application and is an extra effort to be considered under effort estimation.

Prototyping is used to allow the users evaluate developer proposals and try them out before implementation.

It also helps understand the requirements which are user specific and may not have been considered by the developer during product design.

Key Objectives

Software prototyping should only be used when the efforts spent in building the prototype add considerable value to the final software developed.

Benefits

The main advantages of the Rapid Prototyping are as follows:

- Increased user involvement in the product even before its implementation
- Since a working model of the system is displayed, the users get a better understanding of the system being developed
- Reduces time and cost as the defects can be detected much earlier
- Quicker user feedback is available leading to better solutions
- Missing functionality, confusing or difficult functions can be identified easily

Drawbacks

The main disadvantages of the Rapid Prototyping are as follows:

- Risk of insufficient requirements analysis owing to too much dependency on the prototype
- Users may get confused in the prototypes and actual systems
- Practically, this methodology may increase the complexity of the system as scope of the system may expand beyond original plans
- Developers may try to reuse the existing prototypes to build the actual system, even when it is not technically feasible
- The effort invested in building prototypes may be too much if it is not monitored properly

Applicability

Rapid Prototyping is of the greatest value for development of the software having high levels of user interactions, such as online applications.

Systems which need users to fill out forms or go through various screens before data is processed can use prototyping very effectively to give the exact look and feel even before the actual software is developed.

Software that involves too much data processing and most of the functionality is internal with very little user interface does not usually benefit from prototyping.

Prototype development could be an extra overhead in such projects and may need a lot of extra effort.

5.6. Scrum

Scrum is an Agile — lightweight, iterative, and incremental — framework for developing, delivering, and sustaining complex products.

It is designed for teams of ten or fewer members, who break their work into iterations, called sprints, no longer than one month and most commonly two weeks.

The Scrum team tracks progress in 15-minute time-boxed daily meetings, called **Daily Scrums**.

At the end of the Sprint, the team holds Sprint Review, to demonstrate the work done, and Sprint Retrospective to improve continuously.

Scrum Principles

Scrum framework enables teams to self-organize by encouraging physical colocation or close online collaboration of all team members, as well as daily face-to-face communication among all team members and disciplines involved.

A key principle of Scrum is the dual recognition that customers may change their minds about what they want or need — producing requirements volatility — and that there will be unpredictable challenges for which a predictive approach is not suited.

As such, Scrum adopts an evidence-based empirical approach – accepting that the problem cannot be fully understood or defined up front, and instead focusing on how to maximize the team's ability to deliver quickly, to respond to emerging requirements, and to adapt to evolving technologies and changes in market conditions.

VI. Scrum Framework

6. Scrum Framework

Scrum is a feedback-driven empirical approach underpinned by transparency, inspection, and adaptation.

Transparency means that every process, workflow, or progress within the Scrum framework should be visible to those responsible for the outcome.

Frequent Inspection of the product being developed and how well each one performs is the best way to make processes visible for the whole team.

Adaptation, based on the Frequent Inspection, is the ability of the team to spot the initial goal deviations and to adjust the development process.

Scrum Values

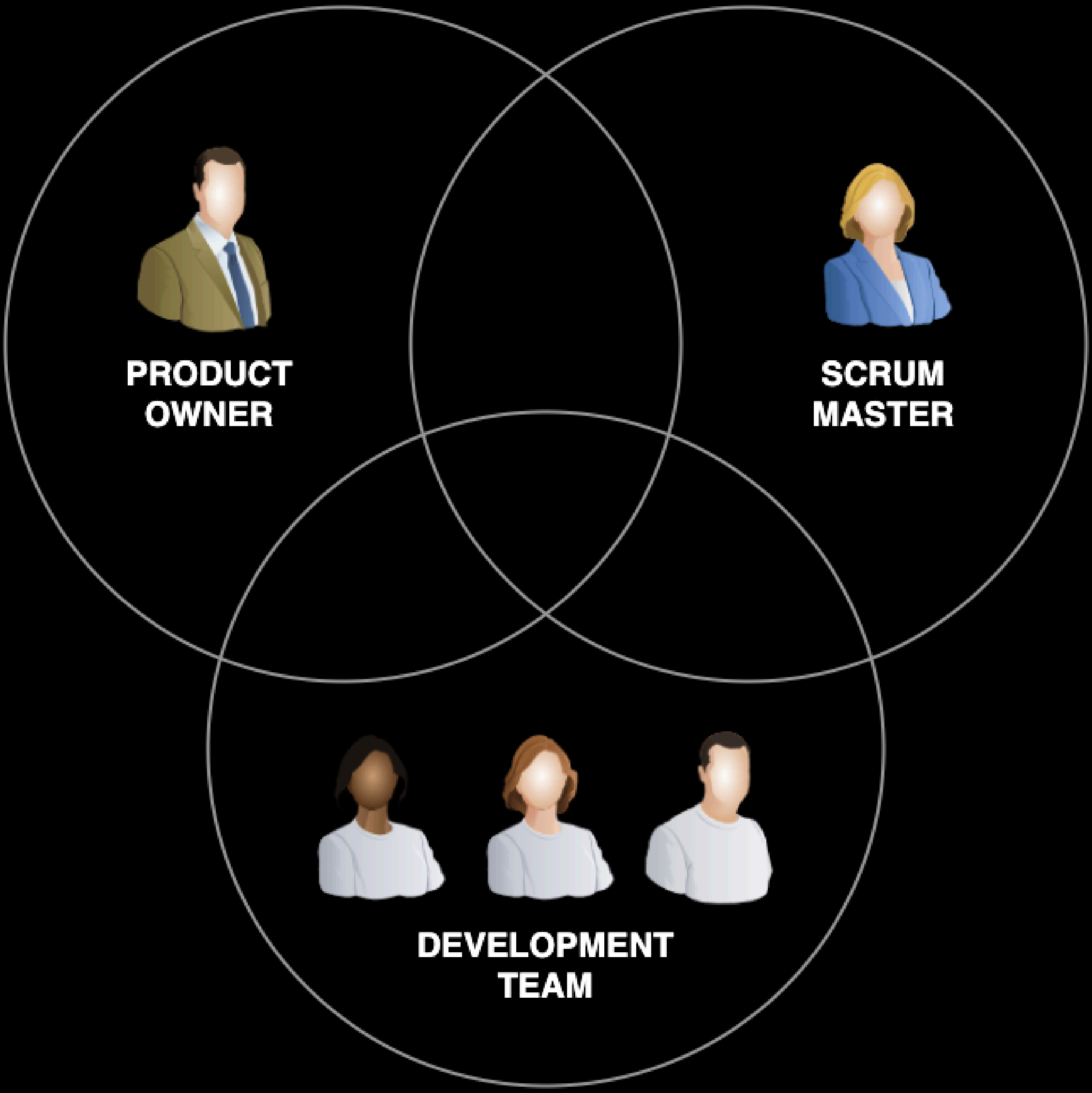
Scrum framework emphasizes five core values that guide the work, actions, and behavior of Scrum Team members:

- Focus — keeps the Team focused on its goal avoiding other work
- Respect — makes members respecting each other as skilled professionals
- Openness — agrees the Team and Stakeholders to be open about their job
- Commitment — helps every member to commit in achieving the common goal
- Courage — allows the Team to do right things facing the tough issues

6.1. Scrum Team

The **Scrum Team** is the fundamental unit of the Scrum framework — a cohesive group of professionals working together to deliver valuable, high-quality outcomes through collaboration and iterative progress.

Scrum Team Structure



A Scrum Team consists of three key roles, each with distinct responsibilities:

- **Product Owner** — Represents stakeholders, defines priorities, and ensures the team delivers maximum business value
- **Scrum Master** — Acts as a facilitator, removing obstacles and fostering an environment where the team can work efficiently
- **Development Team** — A self-organizing, cross-functional group of professionals who design, build, and test the product increment

Core Principles

Scrum thrives on transparency, collaboration, and continuous improvement.

To succeed, the team must:

- Align on a shared goal — Every member contributes toward a unified objective
- Uphold Scrum values — Commitment, courage, focus, openness, and respect
- Communicate frequently — Daily interactions (e.g., stand-ups, sprint reviews) ensure alignment and adaptability

By fostering a culture of trust, accountability, and adaptability, the Scrum Team maximizes efficiency and delivers meaningful results in each sprint.

6.2. Product Owner

Product Owner — PO — is the representative of the product Stakeholders and a Customer responsible for the positive business results.

Product Owner defines the software functionalities in customer-centric terms — typically, User Stories — and prioritizes them according to their importance for the software.

A Scrum Team should have only one Product owner, although a Product owner could support more than one team.

Key Activities

Product Owner is focused on the business side of the software development consolidating efforts of the Stakeholders and the Scrum team.

Product Owner does not indicate how the Scrum team reaches a technical goal, but rather seeks accordance amid the Scrum team members.

The role is crucial and requires a deep understanding of both sides, Business and Developers, within the Scrum team.

Key Objectives

A good product owner should be able to communicate what the business needs, select the best ways to achieve their objectives, and convey the message to all stakeholders including the developers using technical language, as required.

The Product Owner uses Scrum's empirical tools to manage highly complex work, while controlling risk and achieving value.

Key Responsibilities

Communication is a core responsibility of the product owner.

The ability to convey priorities and empathize with team members and stakeholders is vital to steer product development in the right direction.

The product owner role bridges the communication gap between the team and its stakeholders, serving as a proxy for stakeholders to the team and as a team representative to the overall stakeholder community.

Tasks

As the face of the team to the stakeholders, the following are some of the communication tasks of the product owner to the stakeholders:

- Define and announce releases
- Communicate delivery and team status
- Share progress during governance meetings
- Share significant risks, impediments, dependencies, and assumptions — RIDA — with stakeholders
- Negotiate priorities, scope, funding, and schedule
- Ensure that the software outlook is visible, transparent, and clear

6.3. Scrum Master

Scrum Master is the person accountable for removing the holdbacks on the way of the Team to achieve the development goals and deliverables.

Scrum Master ensures the team follows the Scrum framework principles facilitating the key sessions and encouraging the team to improve.

Key Responsibilities

The core responsibilities of a Scrum master include:

- Product Owner assistance
- Team Support and Empowerment
- Coaching the Scrum Team
- Creating High-Value Increments
- Facilitating Scrum Events

Scrum Master may not have people management responsibilities, so the role of Product owner should never be combined with that of the Scrum master.

6.4. Development Team

Development Team is the unit to carry out all the tasks required to build increments of valuable output.

Team members are referred to as **Developers**.

The term Developer refers to anyone who plays a role in the development and support of the system or product, and may include:

- Analysts
- Architects
- Data specialists
- Designers
- Developers
- Researchers
- QA engineers

— and others.

Team Organization

The Development Team is self-organizing.

While no work should come to the team except through the Product owner and the Scrum master should protect the team from too much distraction, the team should still be encouraged to interact directly with Stakeholders and Customers to gain the maximum of understanding and immediacy of feedback.

6.5. Scrum Artifacts

Scrum Artifacts — also spelled as Artefacts — are information that a Scrum Team and Stakeholders use to detail the software being developed, actions to produce it, and the actions performed during the project.

In the Scrum framework, there are three essential Artifacts that play a crucial role in ensuring transparency and facilitating effective collaboration:

- Product backlog
- Sprint backlog
- Increment

These Artifacts provide transparency, guide decision-making, and serve as a basis for adaptation within the Scrum team and its Stakeholders.

Product Backlog

Product Backlog is a dynamic list of items that represent the work needed to build a product.

It includes new features, enhancements, bug fixes, tasks, and other work requirements.

The Product Owner is responsible for maintaining and prioritizing the items in the backlog.

The commitment associated with the Product Backlog is the Product Goal.

Sprint Backlog

Sprint Backlog is a subset of the Product Backlog items selected for development during a specific time-boxed period called Sprint.

The Development team collaboratively decides which items to include in the Sprint Backlog.

The commitment associated with the Sprint Backlog is the Sprint Goal.

Increment

Increment is the sum of the work completed during a Sprint that adds value to the product.

It includes the features, enhancements, and bug fixes that were developed and meet the Definition of Done.

The commitment associated with the Increment is the Definition of Done.

6.6. Product Backlog

Product backlog is a breakdown of work to be done and contains an ordered list of product requirements that a Scrum Team maintains for a product.

The requirements define features, bug fixes, non-functional requirements, etc — whatever must be done to deliver a viable product.

Product backlog and the business value of each Product Backlog Item — PBI — is the responsibility of the Product Owner.

The product owner prioritizes PBIs based on the following considerations:

- Risk
- Business value
- Item's dependencies
- Item's size
- Date needed

Key Activities

Typically, the Product Owner and the Scrum Team work together to develop the breakdown of work. The Product Backlog:

- Captures requests to modify a product — including new features, replacing old features, removing features, and fixing issues

- Ensures the developers have work that maximizes business benefit of the product

Product Backlog evolves as new information surfaces about the product and about its customers, and so later sprints may address new work.

6.7. Sprint Backlog

The **Sprint Backlog** is the list of work the team must address during the next sprint.

The list is derived by the Scrum Team progressively selecting Product Backlog items in priority order from the top of the Product Backlog.

Tasks

The Product Backlog items may be broken down into tasks by the developers.

Tasks on the Sprint Backlog are never assigned — or pushed — to team members by someone else.

Rather team members sign up for — or pull — tasks as needed according to the Backlog priority and their own skills and capacity.

This promotes self-organization of the developers.

Task Board

The Sprint Backlog is the property of the developers, and all included estimates are provided by the developers.

Often an accompanying Task Board is used to see and change the state of the tasks of the current sprint, like To Do, In Progress and Done.

Re-Prioritization

Once a Sprint Backlog is decided, no additional work can be added to the Sprint Backlog except by the Team.

Once a Sprint has been delivered, the Product Backlog is analyzed and re-prioritized if necessary, and the next set of functionality is selected for the next Sprint.

6.8. Increment

Increment is the potentially releasable output of the Sprint that meets the Sprint goal.

Key Activities

Increment is formed from all the completed Sprint Backlog items, integrated with the work of all previous Sprints.

Increment must be complete, according to the Scrum Team's Definition of done — DoD.

Key Objective

Increment should be fully functioning, and in a usable condition regardless of whether the Product Owner decides to actually deploy and use it.

6.9. Scrum Events

Scrum Events — also called **Ceremonies** — are the main activities that occur inside each Sprint iteration to provide structure, encourage collaboration, and drive continuous improvement within the Scrum process.

In the Scrum framework, there are five essential Events that play a crucial role in ensuring transparency, adaptation, and effective collaboration:

- Sprint
- Sprint Planning
- Daily scrum
- Sprint review
- Sprint retrospective

Sprint

Sprint is a fundamental time-boxed period when a Scrum Team collaboratively works to achieve a specific goal.

Sprint serves as the heartbeat of Scrum, where ideas are transformed into tangible value.

It provides a consistent and short iteration for feedback, allowing the team to inspect and adapt both their work processes and the items they're working on.

Sprints have a fixed length, lasting one month or less.

Shorter Sprints generate more learning cycles and limit risk to a smaller time frame.

Sprint Planning

Sprint Planning is a crucial event in the Scrum framework that sets the stage for a productive Sprint by establishing the Sprint goal and the way it will be completed.

Sprint Planning initiates the sprint by defining the work to be accomplished during that Sprint and ensures that the most important items from the Product Backlog are discussed and aligned with the Product Goal.

Daily Scrum

Daily Scrum — also known as **Daily Standup** — is a short every-day Developers' meeting to inspect the progress toward the Sprint goal and adapt the Sprint backlog on necessity.

The aim of the Daily Scrum is to create an actionable plan for the next day’s work facilitating the decisions and highlighting the impediments for removal.

Sprint Review

Sprint Review is the Sprint completion event for the Scrum team to discuss the Increment with Stakeholders and to determine the future amendments.

Sprint Review is the meeting to showcase which Product Backlog items have been done and which are still pending.

The Scrum Team collaborates on what to do next, providing valuable input for subsequent Sprint Planning.

Sprint Retrospective

Sprint retrospective is a meeting for the Scrum Team to estimate how the accomplished Sprint went regarding individuals, interactions, processes, tools, and their Definition of Done.

Developers share what went well during the Sprint, the encountered issues, and how those issues were or were not solved.

Scrum Team identifies the improvement areas to implement enhancements and increase effectiveness during the next Sprint.

6.10. Sprint

Sprint — also known as **Iteration** or **Timebox** — is the basic duration unit of development in Scrum.

The Sprint length is a length agreed and fixed in advance for each Sprint and is normally between one week and one month — with two weeks being the most common.

Workflow

Each Sprint starts with a Sprint Planning event that establishes a Sprint goal and the required Product Backlog items.

The team accepts what they agree is ready and translates this into a Sprint Backlog, with a breakdown of the work required and an estimated forecast for the Sprint goal.

Each Sprint ends with a Sprint Review and Sprint Retrospective, that reviews progress to show to stakeholders and identify lessons and improvements for the next Sprints.

Emphasis

Scrum emphasizes valuable, useful output at the end of the Sprint that is really done — the software that was fully integrated, tested, documented, and is potentially releasable.

Sprint Planning

At the beginning of a Sprint, the Scrum Team holds a Sprint Planning event to:

- Mutually discuss and agree on the scope of work that is intended to be done during the Sprint

- Select Product Backlog items that can be completed in one Sprint
- Prepare a Sprint Backlog that includes the work needed to complete the selected Product Backlog items
- Agree the Sprint goal and a short description they are forecasting to deliver at the end of the Sprint

As the detailed work is elaborated, some Product Backlog items may be split or put back into the product backlog if the team no longer believes they can complete the required work in a single Sprint.

The maximum duration of a Sprint Planning should not exceed 8 hours for a 4 week Sprint.

6.11. Daily Scrum

The **Daily Scrum** is a daily, time-boxed event where the Development Team synchronizes activities, inspects progress toward the Sprint Goal, and adapts the plan for the next 24 hours.

In simple words, it's a daily team huddle to get on the same page, not a detailed status report for a manager.

The main guidelines for the Daily Scrum are as follows:

- All developers should come prepared
- Anyone is welcome, but only developers may contribute
- Only Team decides how to conduct their Daily Scrum

Key Characteristics

The Daily Scrum:

- Should happen at the same time and place every day
- Is limited — timeboxed — to fifteen minutes

No detailed discussions should happen during the Daily Scrum.

During the meeting, each team member typically answers three questions to drive the inspection and synchronization:

- What did I complete yesterday for the Team to achieve the Sprint goal?
- What do I plan to complete today for the Team to achieve the Sprint goal?
- Do I see any impediment on the way for the Team to achieve the Sprint goal?

Key Objective

The primary goal of the Daily Scrum is to ensure the entire team has a clear, shared understanding of the work and can quickly identify any issues that might prevent them from achieving their Sprint Goal.

This creates focus, promotes self-organization, and minimizes the need for other meetings.

After Party

Once the meeting ends, individual members can get together to discuss issues in detail.

Such a meeting is sometimes known as a Breakout Session or an After Party.

6.12. Sprint Review

At the end of a Sprint, the Team holds two events:

- Sprint Review
- Sprint Retrospective

Sprint Review is an event to inspect the outcome of the Sprint and determine future adaptations.

The Scrum Team presents the results of their work to key Stakeholders and progress toward the Product Goal is discussed.

Key Characteristics

During a Sprint review the team:

- Reviews the work that was completed and the planned work that was not completed
- Presents the completed work — Demo — to the Stakeholders and collaborates with the Stakeholders on what to work on next

Sprint Review is a working session and the Scrum Team should avoid limiting it to a presentation.

Guidelines

The guidelines for Sprint Reviews are:

- Incomplete work cannot be demonstrated
- Recommended duration is two hours for a two-week Sprint

6.13. Sprint Retrospective

Sprint Retrospective is a meeting to plan ways to increase quality and effectiveness.

The Scrum Team inspects how the last Sprint went with regards to individuals, interactions, processes, tools, and their Definition of Done.

At the Sprint Retrospective, the Team:

- Reflects on the past Sprint
- Identifies and agrees on continuous process improvement actions

Guidelines

Guidelines for Sprint retrospectives:

- Recommended duration is one-and-a-half hours for a two-week Sprint
- Scrum Master is to facilitate the event

Key Features

Three main questions should be answered during the Sprint Retrospective:

- What went well during the Sprint?
- What went wrong during the Sprint?
- What to improve in the next Sprint?

Limits

Sprint Retrospective is the conclusion of the Sprint.

Thus, its usual maximum is three hours for a month-long Sprint.

The shorter a Sprint, the lesser a timebox.

6.14. Backlog Refinement

Backlog Refinement — also called **Grooming** — is the ongoing process of reviewing Product Backlog items and checking they are appropriately prepared and ordered in a way that makes them clear and executable for teams once they enter Sprints via the Sprint Planning activity.

During the Grooming:

- Product Backlog items may be broken into multiple smaller ones
- Acceptance criteria may be clarified
- Dependencies may be identified and investigated

Although not originally a core Scrum practice, Backlog Refinement has been added to the Scrum Guide and adopted as a way of managing the quality of Product Backlog items entering a Sprint, with a recommended investment of up to 10% of a Team's Sprint capacity.

Technical Debt

Technical Debt — also known as **Code Debt** — is the implied cost of future rework caused by choosing quick, easy solutions now instead of better approaches that would take longer.

Technical Debt may also be discussed during the Backlog Refinement.

Cancelling a Sprint

The Product Owner can cancel a Sprint if necessary.

The Product Owner may do so with input from the Team, Scrum Master or management.

For instance, management may wish the Product Owner to cancel a Sprint if external circumstances negate the value of the Sprint goal.

If a Sprint is abnormally terminated, the next step is to conduct a new Sprint Planning where the reason for the termination is reviewed.

6.15. Scrum Workflow

Scrum is an agile framework designed to deliver high-quality software iteratively and incrementally.

Its structured yet adaptive workflow emphasizes:

- Flexibility
- Collaboration
- Continuous improvement

and may be represented using the following scheme.

Key Stages of the Scrum Process

1. Product Backlog Creation and Refinement

- The Product Owner maintains a prioritized list of features, enhancements, and fixes called the Product Backlog
- Items — user stories, bugs, tasks — are refined with estimates and acceptance criteria

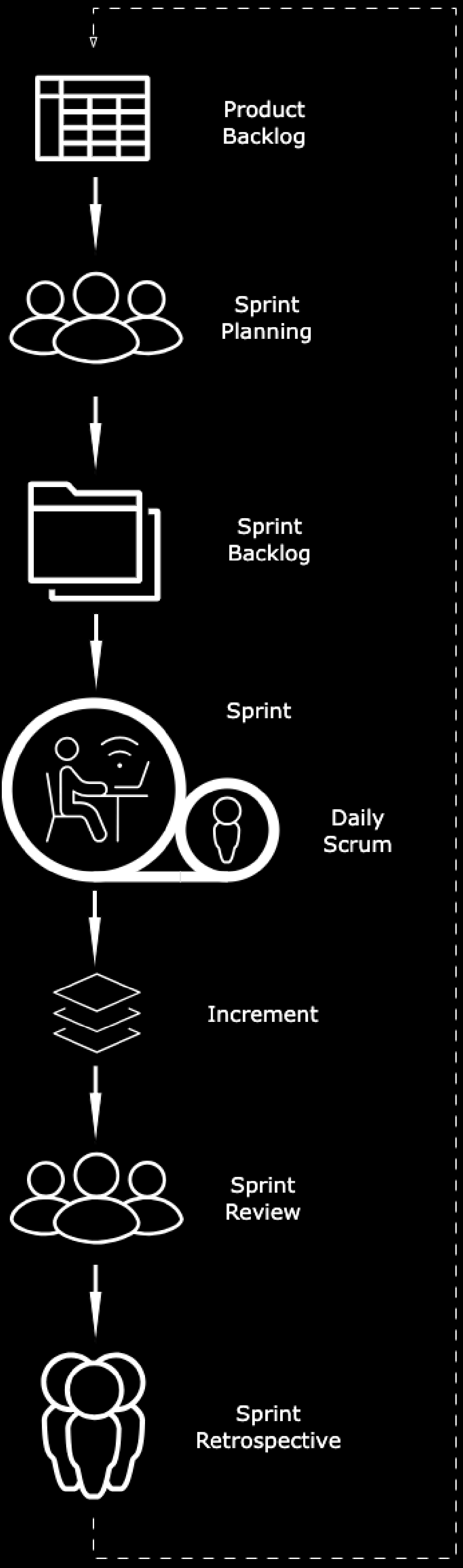
2. Sprint Planning

- The Team selects a set of backlog items for the upcoming Sprint
- Defines the Sprint Goal and creates a Sprint Backlog

3. Daily Scrum

A 15-minute daily meeting for the team to:

- Synchronize team members' work activities
- Ensure transparency and quick issue resolution



Scrum Process

4. Sprint Execution

- The Development Team builds, tests, and integrates features in short cycles
- Work is tracked via a Sprint Board with its To Do, In Progress, and Done

5. Sprint Review

- At the end of the Sprint, the team demonstrates the working product increment to stakeholders
- Feedback is collected to adjust priorities in the Product Backlog

6. Sprint Retrospective

- The team reflects on what went well, what didn't, and how to improve in the next Sprint
- Focuses on process improvements — tools, communication, etc

Core Scrum Artifacts

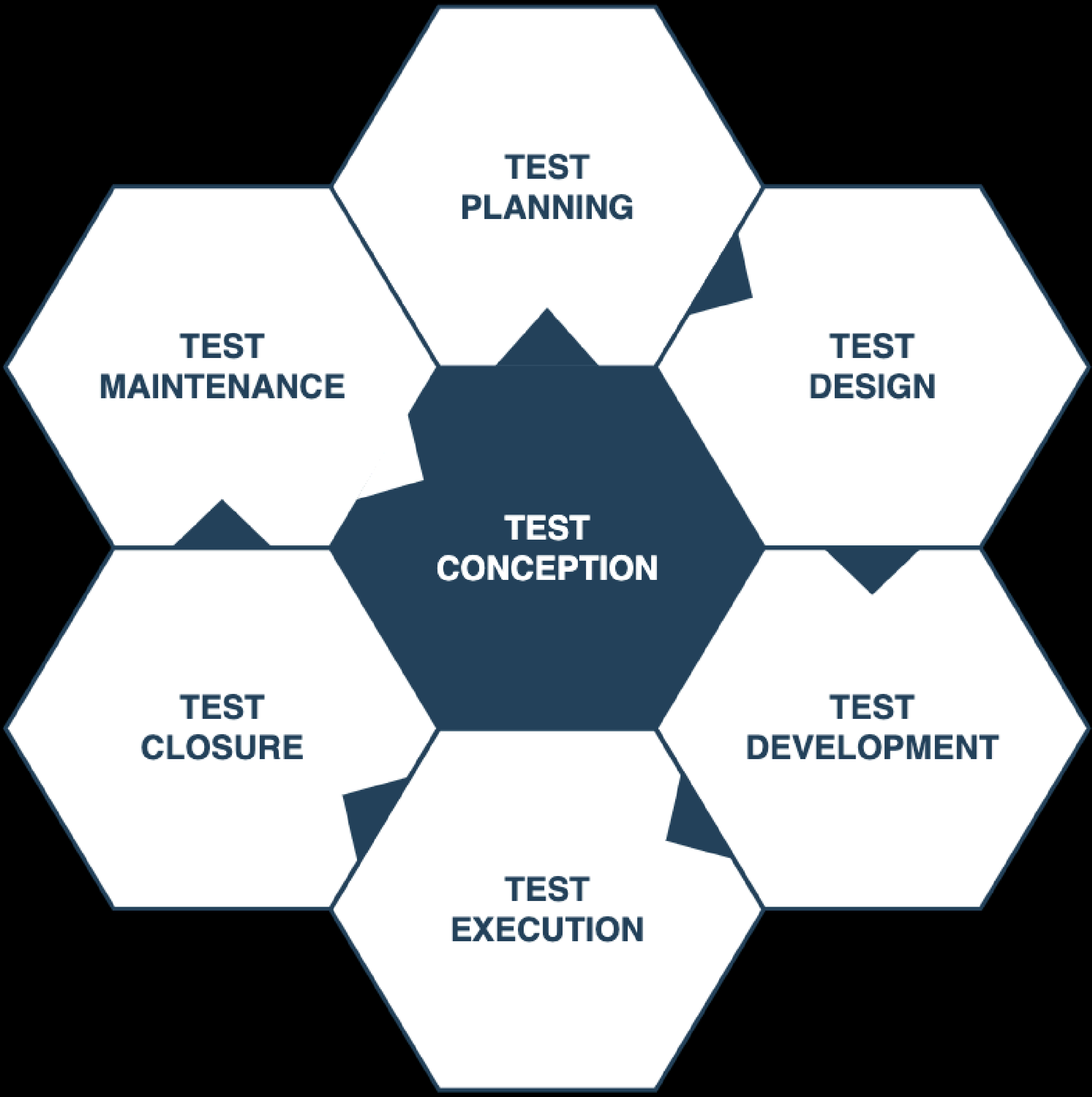
The core Scrum Artifacts are:

- Product Backlog — Dynamic wishlist of all desired features.
- Sprint Backlog — Subset of backlog items committed to in a Sprint.
- Increment — Shippable product version after each Sprint.

VII. Software Testing Life Cycle

7. Software Testing Life Cycle

Software Testing Life Cycle — STLC — is a sequence of verification and validation activities conducted during SDLC to ensure software quality goals are met.



Software Testing Life Cycle

STLC Stages

Software testing life cycle consists of the following methodological stages to certify a software product:

1. Test Conception
2. Test Planning
3. Test Design
4. Test Development
5. Test Execution
6. Test Closure
7. Test Maintenance

Each of these stages has a definite entry and exit criteria, activities, and deliverables associated with it.

7.1. Test Conception Stage

Test Conception is the stage to identify the scope of testing and estimate if the software requirements — functional and operational — are testable.

The Test Conception stage is also to assess the possibility of test automation.

Key Activities

The Test Conception activities are usually aimed to:

- Identify types of tests to be created
- Gather details about testing priorities
- Prepare Requirements Traceability Matrix — RTM
- Identify the supposed test environment
- Analyze automation feasibilities

Key Deliverables

The primary deliverables of the Test Conception stage commonly include:

- Requirements Traceability Matrix — RTM
- Automation Feasibility Report — AFR

7.2. Test Planning Stage

Test Planning is the stage to identify the activities and resources necessary to meet the testing goals.

Test Planning is commonly performed according to the Software Requirement Specification, Test Strategy, and Risk Analysis.

Test Planning is also the stage to identify testing metrics and ways to track them.

Test Plan

Test Plan is a formal document that describes the scope, approach, resources, and schedule of intended testing activities.

It serves as a blueprint for the testing process, identifying what will be tested, how it will be tested, who will do the testing, and when it will be tested.

The main components of the Test Plan are:

- Testing objectives
- Testing scope
- Testing risks
- Test coverage
- Required resources
- Team roles
- Testing tools
- Testing schedule
- Test deliverables

— and other values.

Key Deliverables

The main deliverables of the Test Planning stage are:

- Test Plan
- Testing Schedule

— where the **Testing Schedule** is a detailed timeline that specifies when each testing activity will occur, how long it will take, what resources are needed, and the sequence of testing tasks.

7.3. Test Design Stage

Test Design is the stage for the Checklists and Test Cases to be created, reviewed, and updated according to the Test Plan.

The Test Design stage may and should actually start earlier than the very process of software development.

This stage ensures systematic and efficient testing by defining what to test, how to test, and what data to use.

Key Activities

The Test Design stage is designated to create:

- Checklists based on Exploratory testing
- Test Cases based on extended Checklists
- Test Scripts aimed to automate Test Cases
- Test Data necessary for the Artifacts above

Key Deliverables

The main deliverables of the Test Design stage are:

- Checklists
- Test Cases
- Test Scripts
- Test Data

7.4. Test Development Stage

Test Development is the stage where the testing team designs, creates, and prepares all necessary artifacts needed to execute tests.

These artifacts — Test Cases, Test Scripts, Test Data — are followed through during the Test Execution stage to ensure that the software behaves as expected.

Key Objectives

The primary objectives of the Test Development stage are:

- Define test coverage — ensure all requirements are tested
- Create reusable test cases — for current and regression testing
- Prepare test data — inputs, databases, APIs
- Automate test scripts — where automation is applicable
- Ensure traceability — link test cases to requirements

The stage bridges Test Design and Test Execution stages ensuring structured, repeatable, and efficient validation of software.

By investing in well-designed Test Cases, Test Data, and Test automation teams can:

- Reduce defects in production
- Accelerate regression testing
- Improve audit compliance

Key Activities

During this stage, the testing team focuses on the following core activities:

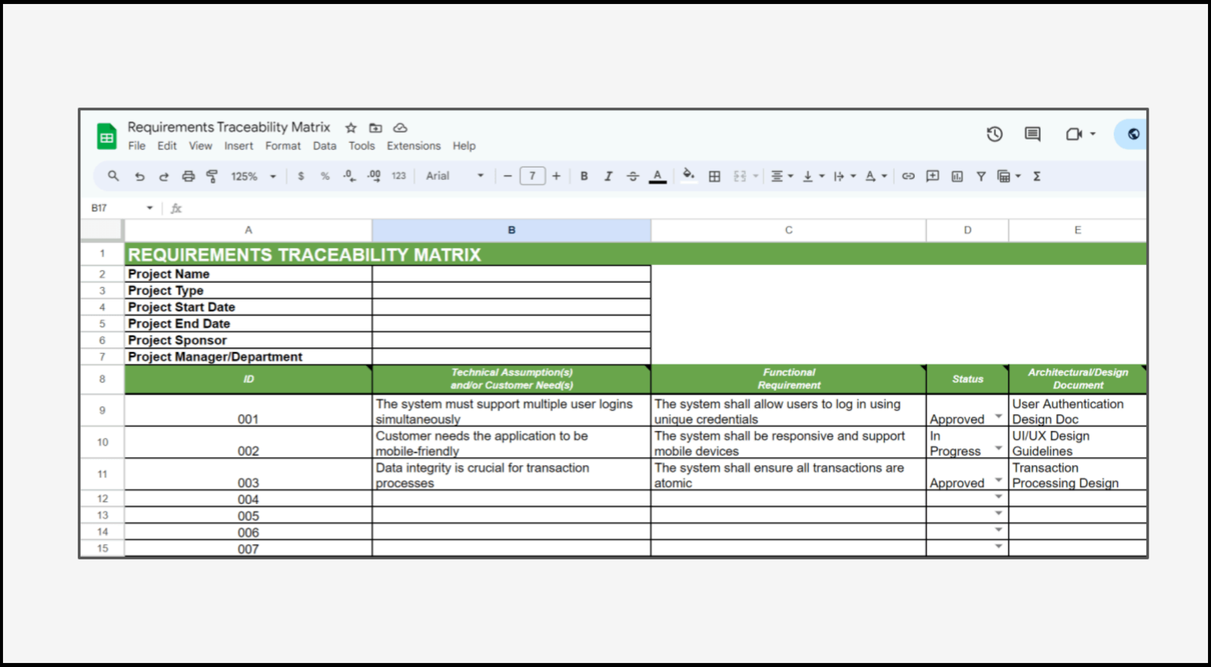
- Test Case Design — Creating detailed, step-by-step procedures to validate specific requirements

- Test Data Preparation — Generating and managing the data needed to execute the test cases
- Test Script Development — Writing automated scripts for regression or performance testing
- Test Environment Setup — Preparing and configuring the hardware/software environment where tests will run

Key Deliverables

The main deliverables of the Test Development stage are:

- Test Cases — Step-by-step validation procedures
- Test Data — Inputs, databases, and environment setups
- Automation Scripts — Code for automated test execution
- Traceability Matrix — Ensures all requirements are tested
- Test Suite Summary — Overview of the test coverage


Requirements Traceability Matrix

REQUIREMENTS TRACEABILITY MATRIX

Project Name	
Project Type	
Project Start Date	
Project End Date	
Project Sponsor	
Project Manager/Department	

ID	Technical Assumption(s) and/or Customer Need(s)	Functional Requirement	Status	Architectural/Design Document
001	The system must support multiple user logins simultaneously	The system shall allow users to log in using unique credentials	Approved	User Authentication Design Doc
002	Customer needs the application to be mobile-friendly	The system shall be responsive and support mobile devices	In Progress	UI/UX Design Guidelines
003	Data integrity is crucial for transaction processes	The system shall ensure all transactions are atomic	Approved	Transaction Processing Design
004				
005				
006				
007				

Requirements Traceability Matrix

7.5. Test Execution Stage

Test Execution is the stage where the QA team runs the developed Test Cases and Test Scripts on the actual software builds according to the Test Plan.

Key Activities

Test Execution is the "hands-on" stage where the software is actively validated against its requirements and commonly includes:

- Environment deployment and setup
- Test cases and test scripts execution
- Test cases and test scripts adjustment
- Defect reporting
- Defect retesting

When bugs are fixed, the developer team renders a new build for the Quality Assurance team to retest software.

Key Deliverables

The main Test execution deliverables are:

- Requirements Traceability Matrix mapped with:
    - Bugs
    - Test cases
    - Test scripts
- Test Cases adjusted to the current application state
- Test Scripts updated to the current application version
- Defect Reports

7.6. Test Closure Stage

Test Closure is the stage when test execution activities of the current development cycle are formally concluded.

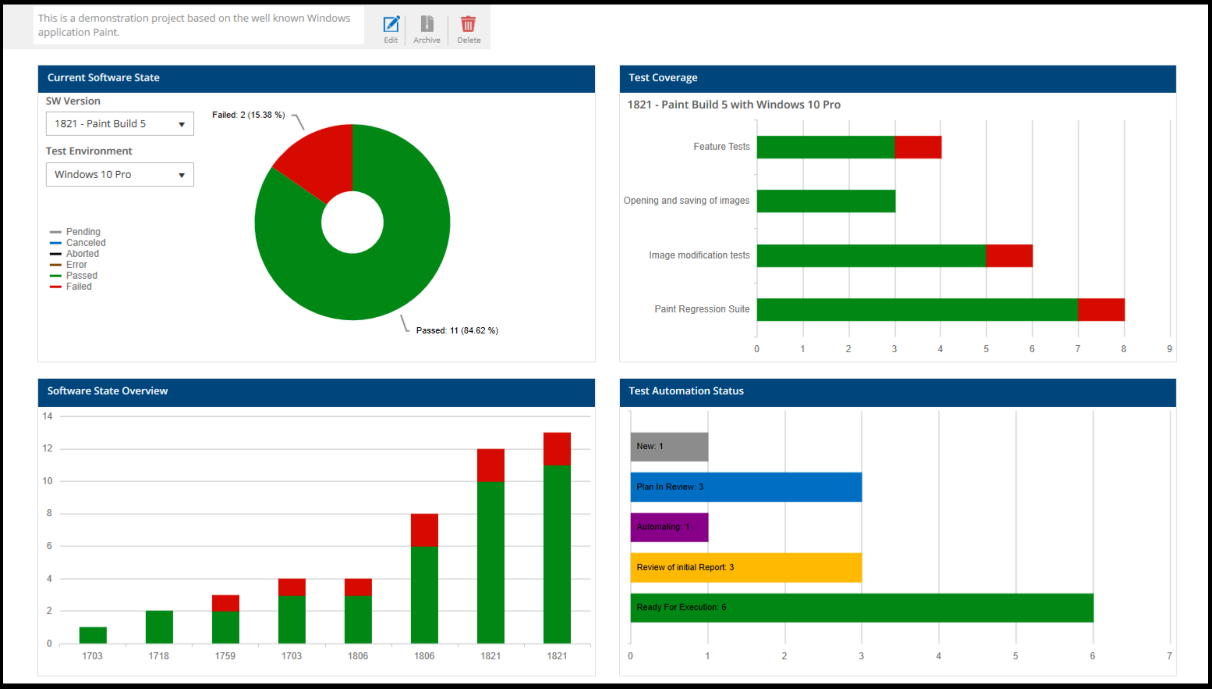
Key Objectives

The main purpose of the Test Closure is to evaluate and assess the overall effectiveness and efficiency of the Test Execution stage.

Test Closure allows also to document the executed tests outcomes gathering them in Test Results Report — TRR — and to store the Test Artifacts keeping them for future reference.

Test Results Report

Test Results Report — or **Test Completion Report** — is the main document of the Test Closure stage which provides insights into discovered and resolved issues.



This is a demonstration project based on the well known Windows application Paint.

SW Version: 1821 - Paint Build 5

Test Environment: Windows 10 Pro

Test Results Report

Test Results Report commonly includes:

- Consolidated Test Results
- Detailed Error Analysis
- Metrics Presentation

Test Results Report signals the completion of testing activities and informs Stakeholders about the cutoff of the Testing stage.

Key Deliverables

Test Closure deliverables commonly include:

- Test Status Report
- Test Results report
- Test metrics

In summary, Test Closure ensures that testing objectives are met, errors are documented, and the Testing stage successfully concludes.

7.7. Test Maintenance

Test Maintenance is the ongoing stage where test assets — cases, scripts, data, and environments — are updated, optimized, and retired to keep pace with evolving software.

Unlike one-time test execution, Test Maintenance ensures long-term relevance, efficiency, and accuracy of testing processes as the application changes.

Key Objectives

The main goals of the Test Maintenance stage include:

- Keeping tests healthy — Test maintenance ensures that both manual test cases and automated test scripts remain relevant and effective as the application evolves.
- Automation framework continuity — Automation framework dependencies are to be aligned with the changes to the tools or third-party libraries.
- Regression testing — Helps ensure that the code changes do not break existing functionality, and automated regression testing suite typically contains a big number of tests, which require ongoing maintenance to validate the application properly.
- Continuous Integration — Automated tests run through Continuous Integration pipelines to identify and resolve issues quickly; ensuring that tests are always up to date is crucial in this context, as builds won't complete if tests fail.
- Reporting — Regular reporting helps Quality Assurance Engineers identify broken tests that need updating.

Key Activities

- Test Artifact Review and Updates:
  - Test Cases — Add or remove steps based on feature changes and re-prioritize test cases
  - Test Data — Refresh datasets to match production changes and anonymize sensitive data for compliance
- Automated Test Script Maintenance:
  - Fix broken locators after UI redesigns
  - Refactor scripts
  - Update libraries, frameworks, and other dependencies
- Test Suite Optimization:
  - Remove redundancy — Merge duplicate tests
  - Improve coverage — Add tests for untested scenarios
  - Tag tests — smoke, regression, sanity — for better execution control
- Traceability Matrix Updates
  - Ensure test cases map to current requirements
  - Highlight gaps where new tests are needed

Key Deliverables

The main deliverables of the Test Maintenance stage are:

- Updated Test Cases — Aligned with new software functionality
- Refactored Test Scripts — Improved in stability and performance
- Test audit report — Reflecting the added or removed tests
- Flaky test log — Documentation of unstable tests and fixes

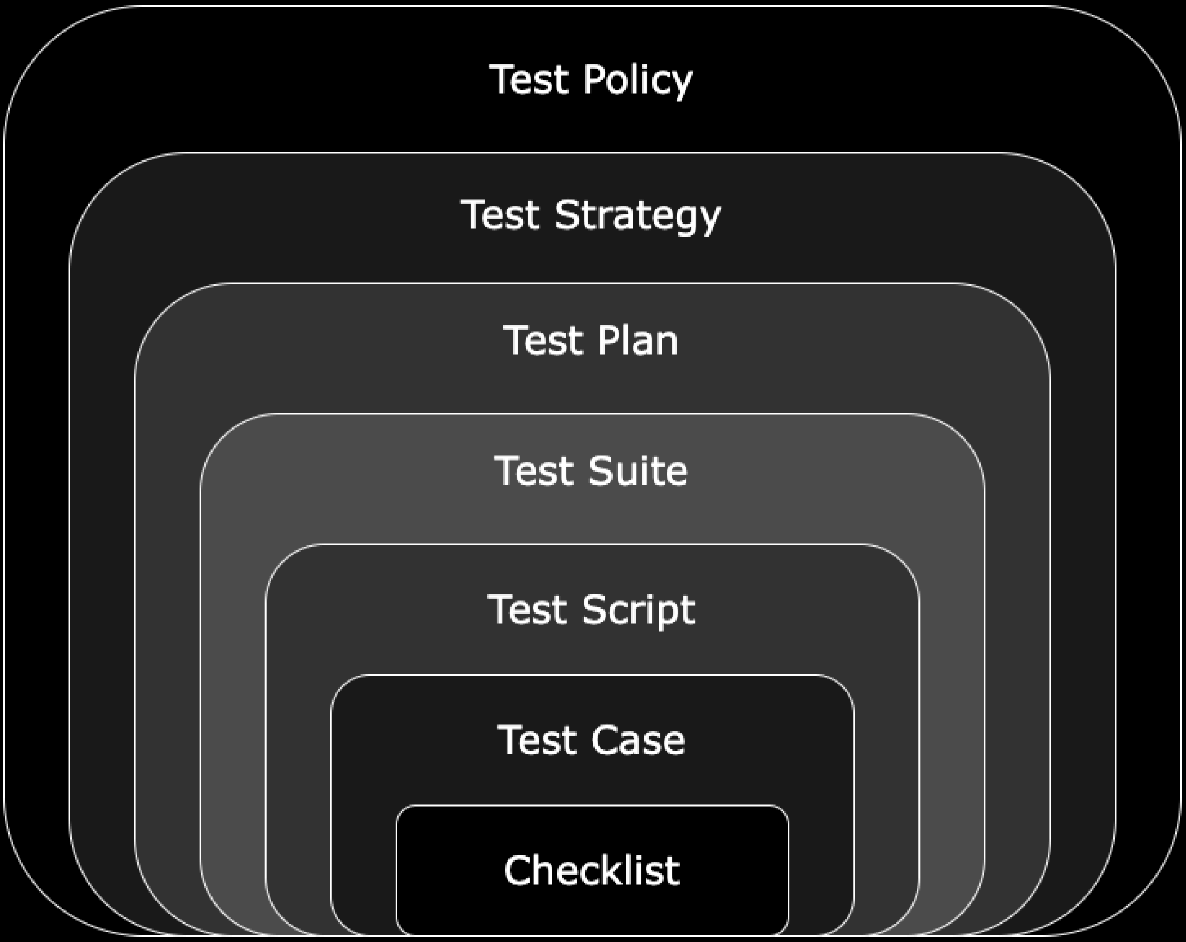
VIII. Test Documentation

8. Test Documentation

Test Documentation is a set of test documents used for the purpose of Software quality assurance.

Test Documentation Scopes

The Test Documentation scopes may be represented the following way.



Test Documentation Scopes

Documentation Categories

Test Documentation may be split into three main categories:

- Test Execution documentation
- Test Automation scripts and data
- Test Conception documentation

Test Execution Documentation

The main Test execution documents are:

- Checklist
- Test Case — or Test Scenario
- Test Suite

Test case being the most important document for the SQA.

Test Automation Documents

The main test automation documents include:

- Test Script
- Test Data

In automation, often, test data is being created automatically.

Though Test data is used in manual testing either, in automation it plays a special role and meaning.

Test Policy Documentation

The main Test conception documentation include:

- Test Plan
- Test Strategy
- Test Policy

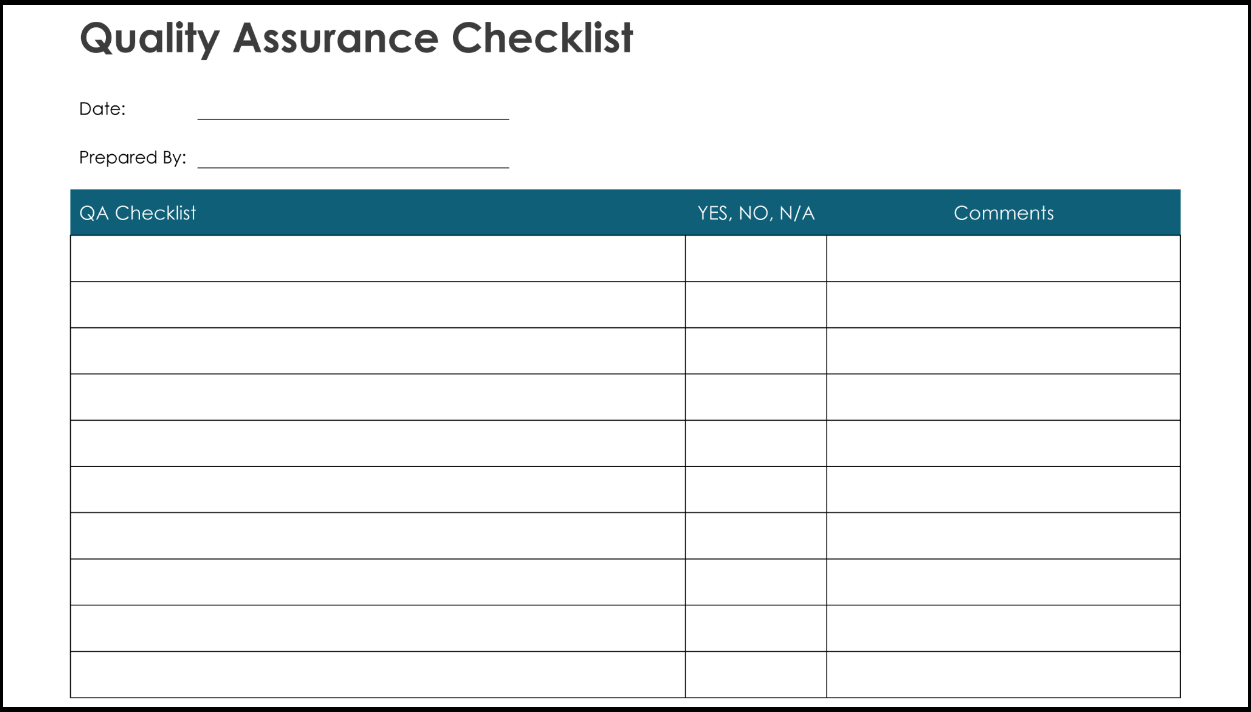
Test Plan and Test Strategy may either be separate documents or a single one; the choice depends on the documentation developer and the software scope.

If a project is bigger in size, it makes sense to have different documents — else, these documents may be united in one.

8.1. Checklist

Checklist is a flat list of verifications to undertake during the Testing stage.

It helps to split the software functionality into separate testing blocks and allows to understand the scope of testing and how many checks failed.



Quality Assurance Checklist

Date: ____________________

Prepared By: ____________________

QA Checklist	YES, NO, N/A	Comments

In Checklist, the test order may be random because it does not matter.

Checklists are common for initial stages of the project when Test Cases are only planned for the future development.

Benefits

The main advantages of Checklists are as follows:

- Flexibility — Checklists can be used in all testing types
- Simplicity — Checklists are easy to create and maintain

- Quickness — Checklists are fast to develop and understand
- Brevity — Checklists have only a couple of fields to fill out
- Results analyzability — Checklists are easy to follow and examine
- Team integration — Checklists simplify QA members onboarding
- Deadlines control — Checklists let control test accomplishment

Drawbacks

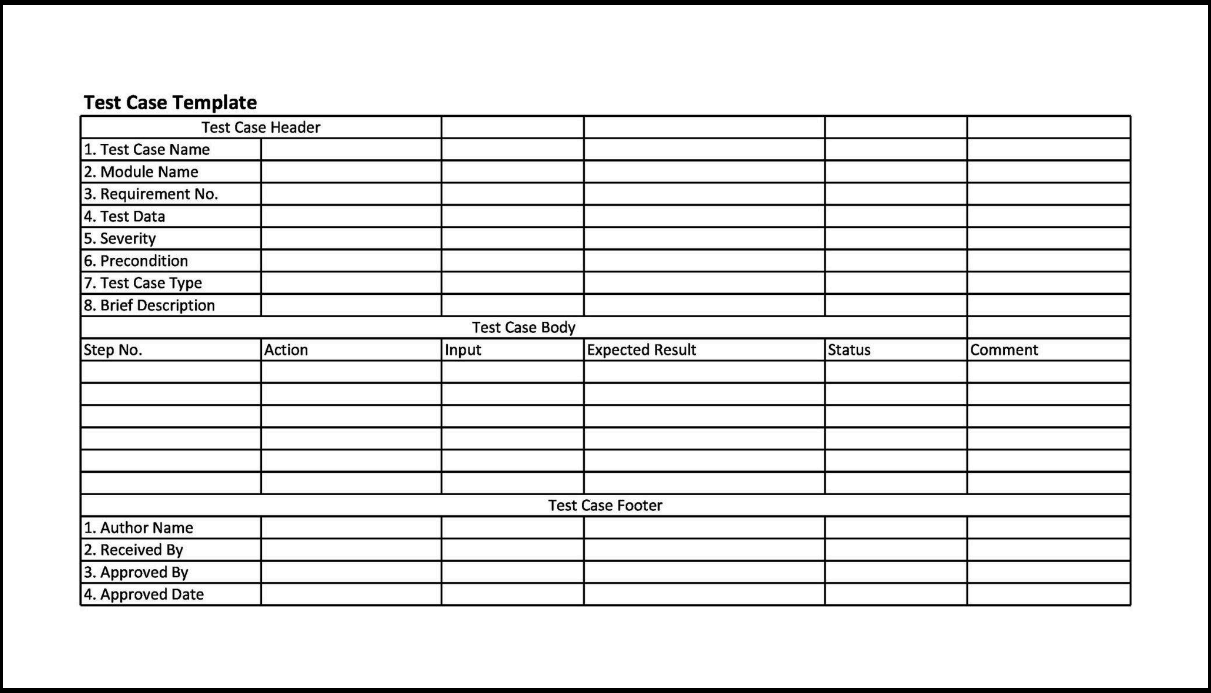
The main disadvantages of using Checklists in testing are:

- Different Interpretation — QA Engineers can accomplish identical tasks using different approaches
- Coverage Gaps — It is difficult to capture all functional or structural components, especially those of higher levels
- Item Overlap — Trying to cover a big scope of material may lead to duplicated tasks and, as a consequence, to excessive testing
- Reporting Problems — Checklists can hardly describe complex system components, functions, and their interaction

8.2. Test Case or Test Scenario

Test Case — often called a **Test Scenario** — is a sequence of steps dedicated to verify the expected software behavior.

The cardinal difference between a Checklist and a Test Case is the Expected Result field obvious for the Test Cases.



Test Case Template

Test Case Header					
1. Test Case Name					
2. Module Name					
3. Requirement No.					
4. Test Data					
5. Severity					
6. Precondition					
7. Test Case Type					
8. Brief Description					
Test Case Body					
Step No.	Action	Input	Expected Result	Status	Comment
Test Case Footer					
1. Author Name					
2. Received By					
3. Approved By					
4. Approved Date					

Test Case is the main testing document and is often synonymous for the very term Test.

Test Case Types

There are two types of Test cases:

- Informal
- Formal

Informal Test Case is the Test Case used for the software which doesn't have formal requirements being based on the accepted normal operation of the software of a similar class.

Formal Test Case is the Test Case characterized by a known input and expected output, which is figured out before the test is executed.

Mandatory Fields

Formal Test Case format may include various parameters but the mandatory fields are as follows:

- ID
- Description
- Steps
- Expected result
- Status
- Priority

The shortest Test case format may omit the Priority field.

Requirement Traceability Matrix

In order to fully verify that all requirements for the software are met, there must be at least two Test Cases for each requirement:

- Positive
- Negative

If a requirement has sub-requirements, each sub-requirement must have at least two Test Cases either.

Keeping track of the link between the requirement and the test is frequently done using a Requirement Traceability Matrix.

Formal Test Case Structure

Written Test Cases should include a description of the functionality to be tested, and the preparation required to ensure that the test can be conducted.

Typical written Test Case format may include:

- ID — Unique identifier of the Test Case
- Author — Test Case's developer name
- Test Category — The group a Test Case belongs to
- Description — or Summary — The objective of the Test Case
- Pre-conditions — or Pre-requisites — The conditions to be met before the test steps execution
- Test Data — The variables and their values in the Test Case
- Steps — Actions to be performed during the Test Case execution
- Expected Result — The expected outcome of the executed Step
- Post-conditions — The result of the Step execution
- Actual Result — The result retrieved after the Step execution
- Status — The Pass-or-Fail outcome of the Expected and Actual Results comparison
- Priority — The Test Case's importance for Regression Testing
- Automation — A mark whether the Test Case is to be automated
- Automated — A mark whether the Test Case is automated
- Remarks — A set of notes related to the test Step

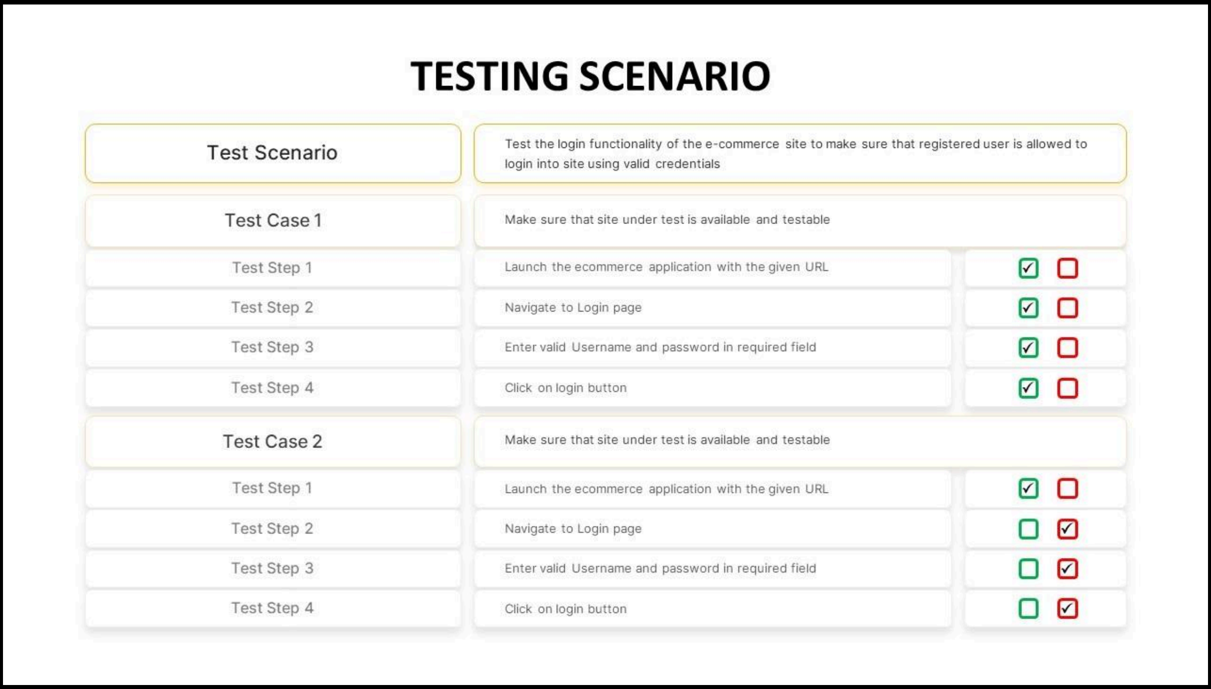
The majority of the fields above are optional and may be omitted in the Test Case format.

Test Scenario

Test Scenario is a step by step end-user activity documented for a certain functionality to be tested.

While the Test Case is better suited for Unit Testing, the Test Scenario better fits for the End-To-End — E2E — Functional Testing.

Test Cases are usually brief verifications while Test Scenarios cover a significant number of steps to validate an expected result and are better suited for automation.



TESTING SCENARIO

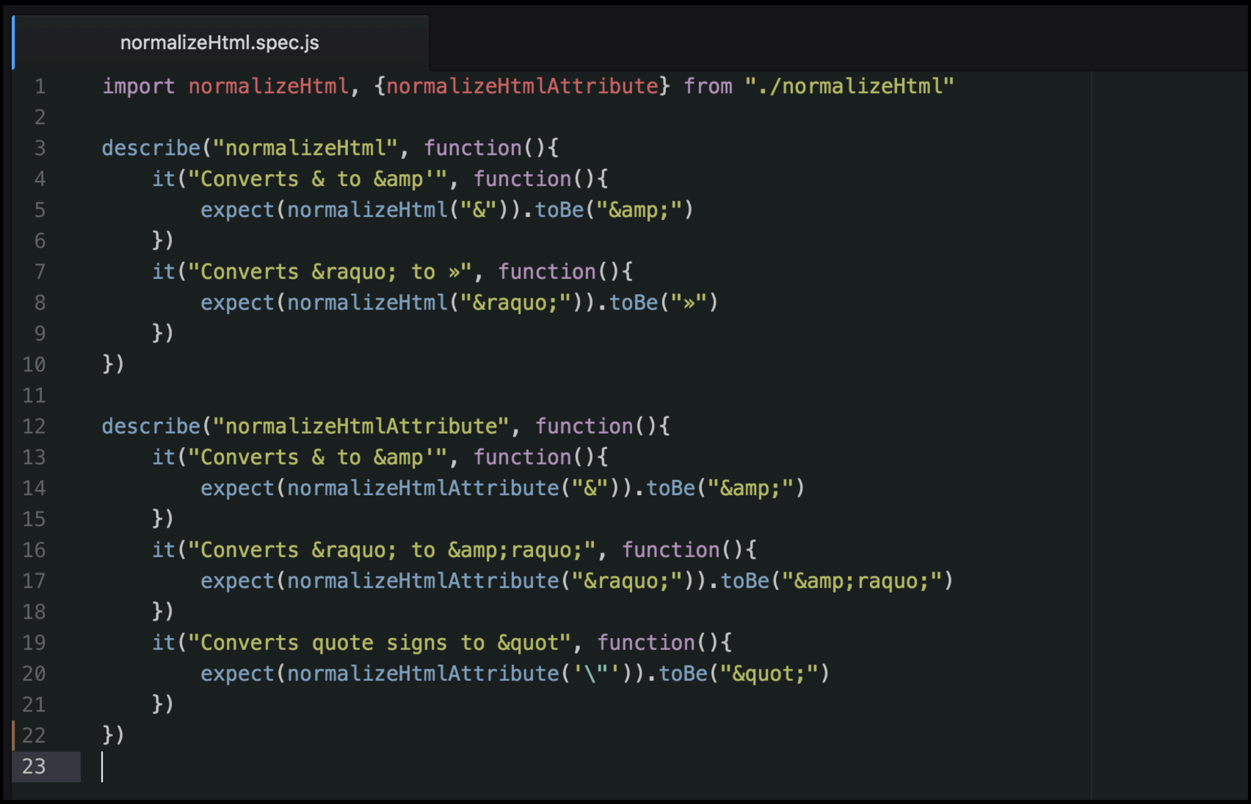
Test Scenario	Test the login functionality of the e-commerce site to make sure that registered user is allowed to login into site using valid credentials		
Test Case 1	Make sure that site under test is available and testable		
Test Step 1	Launch the ecommerce application with the given URL	☑	☐
Test Step 2	Navigate to Login page	☑	☐
Test Step 3	Enter valid Username and password in required field	☑	☐
Test Step 4	Click on login button	☑	☐
Test Case 2	Make sure that site under test is available and testable		
Test Step 1	Launch the ecommerce application with the given URL	☑	☐
Test Step 2	Navigate to Login page	☐	☑
Test Step 3	Enter valid Username and password in required field	☐	☑
Test Step 4	Click on login button	☐	☑

The purpose of the Test Scenario is to follow the end-to-end consequence of steps to test a specific complex issue or troublesome use case.

8.3. Test Script and Test Data

Test Script is a program designated to automate Test Case steps and verifications.

Each Test Script is typically associated with a Test Case.



```
import normalizeHtml, {normalizeHtmlAttribute} from "./normalizeHtml"

describe("normalizeHtml", function(){
    it("Converts & to &amp'", function(){
        expect(normalizeHtml("&")).toBe("&amp;")
    })
    it("Converts &raquo; to »", function(){
        expect(normalizeHtml("&raquo;")).toBe("»")
    })
})

describe("normalizeHtmlAttribute", function(){
    it("Converts & to &amp'", function(){
        expect(normalizeHtmlAttribute("&")).toBe("&amp;")
    })
    it("Converts &raquo; to &amp;raquo;", function(){
        expect(normalizeHtmlAttribute("&raquo;")).toBe("&amp;raquo;")
    })
    it("Converts quote signs to &quot", function(){
        expect(normalizeHtmlAttribute('\"')).toBe("&quot;")
    })
})
```

After the Test Script is implemented, it usually replaces the Test Case associated with it during the Software testing life cycle.

Test Script is commonly created to test a part of the software system functionality.

Test Scripts can be created using general purpose programming languages, special test-script programming languages, or Graphic User Interface test-recording tools.

Test Scripts are crucial while mocking a situation inimitable for a human being, for instance, as a part of Load Testing.

Test Script Components

The main Test Script components are as follows:

- Test Steps — The detailed instructions for each test action
- Expected Results — The behavior expected from the software
- Test Data — The input values needed for the test
- Assertions — The conditions to comply during the script run

Test Script Development

The Test Script development process usually includes:

- Test Case analysis — Understanding of software requirements
- Test Case design — The detailed Test Case creation
- Test Script coding — The Test Case into Test Script translation
- Test Data preparation — The relevant data generation
- Test Script execution — Running the Test Script
- Test Script storage — Keeping the test code in repository

Benefits

Automated testing is advantageous for a number of reasons:

- Test Scripts may run continuously
- Test Scripts do not need a human
- Test Scripts are easily repeatable
- Test Scripts are much faster

Drawbacks

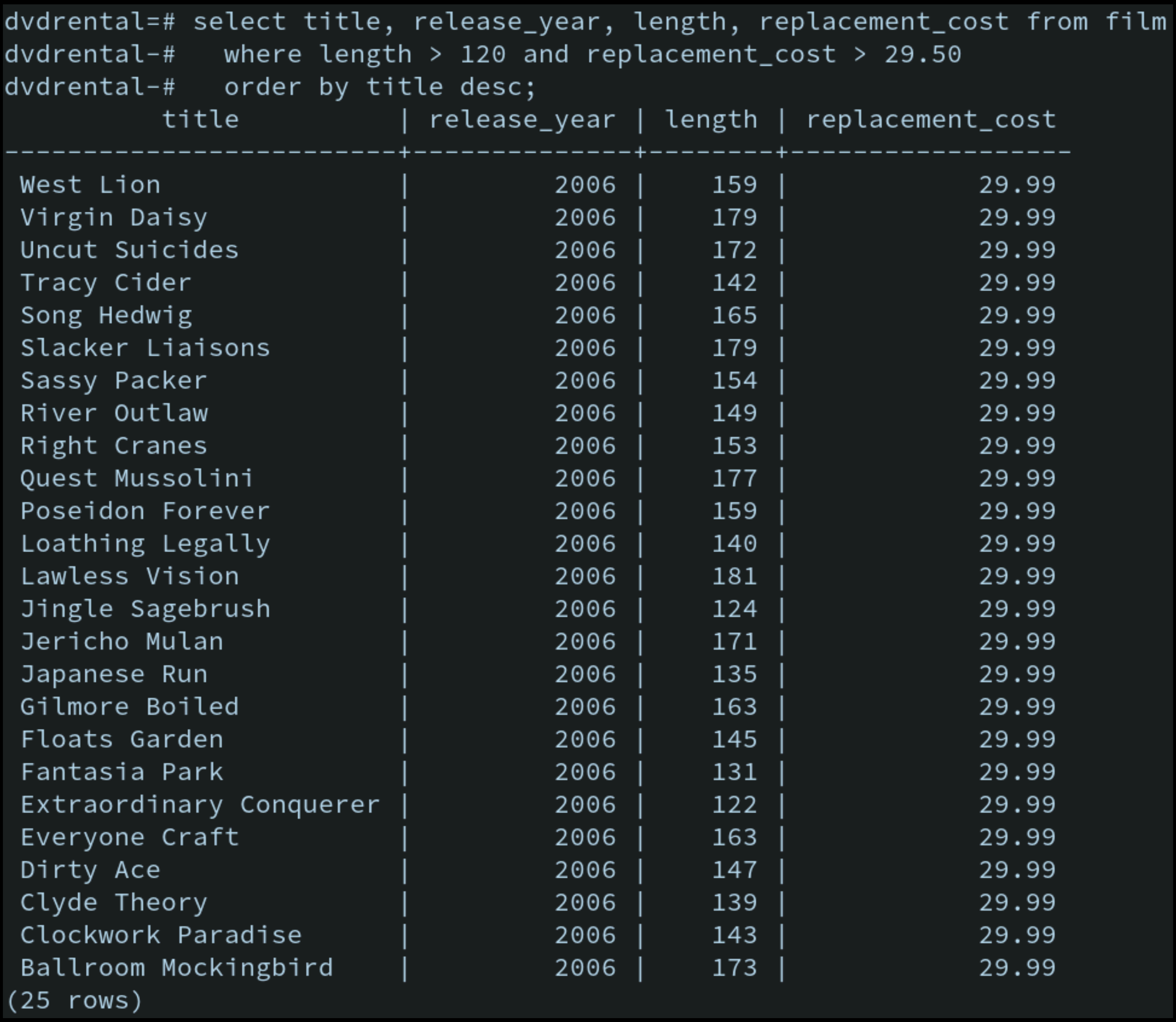
Test Script, as a software to test a software:

- May contain their own flaws
- Require higher level of expertise
- Can only examine what they are programmed to examine

Test Data

Test Data is the data specifically designated to be used in tests.

Test Data may be produced by the tester, or by a program that aids the tester.



```
dvdrental=# select title, release_year, length, replacement_cost from film
dvdrental-#   where length > 120 and replacement_cost > 29.50
dvdrental-#   order by title desc;
          title          | release_year | length | replacement_cost
-------------------------+--------------+--------+------------------
 West Lion               |         2006 |    159 |            29.99
 Virgin Daisy            |         2006 |    179 |            29.99
 Uncut Suicides          |         2006 |    172 |            29.99
 Tracy Cider             |         2006 |    142 |            29.99
 Song Hedwig             |         2006 |    165 |            29.99
 Slacker Liaisons        |         2006 |    179 |            29.99
 Sassy Packer            |         2006 |    154 |            29.99
 River Outlaw            |         2006 |    149 |            29.99
 Right Cranes            |         2006 |    153 |            29.99
 Quest Mussolini         |         2006 |    177 |            29.99
 Poseidon Forever        |         2006 |    159 |            29.99
 Loathing Legally        |         2006 |    140 |            29.99
 Lawless Vision          |         2006 |    181 |            29.99
 Jingle Sagebrush        |         2006 |    124 |            29.99
 Jericho Mulan           |         2006 |    171 |            29.99
 Japanese Run            |         2006 |    135 |            29.99
 Gilmore Boiled          |         2006 |    163 |            29.99
 Floats Garden           |         2006 |    145 |            29.99
 Fantasia Park           |         2006 |    131 |            29.99
 Extraordinary Conquerer |         2006 |    122 |            29.99
 Everyone Craft          |         2006 |    163 |            29.99
 Dirty Ace               |         2006 |    147 |            29.99
 Clyde Theory            |         2006 |    139 |            29.99
 Clockwork Paradise      |         2006 |    143 |            29.99
 Ballroom Mockingbird    |         2006 |    173 |            29.99
(25 rows)
```

Test Data may be recorded for re-use, or used once and then forgotten.

Test Data may be split into two following types:

- Synthetic Data — also called Fake Data
- Representative Data — also called Real Data

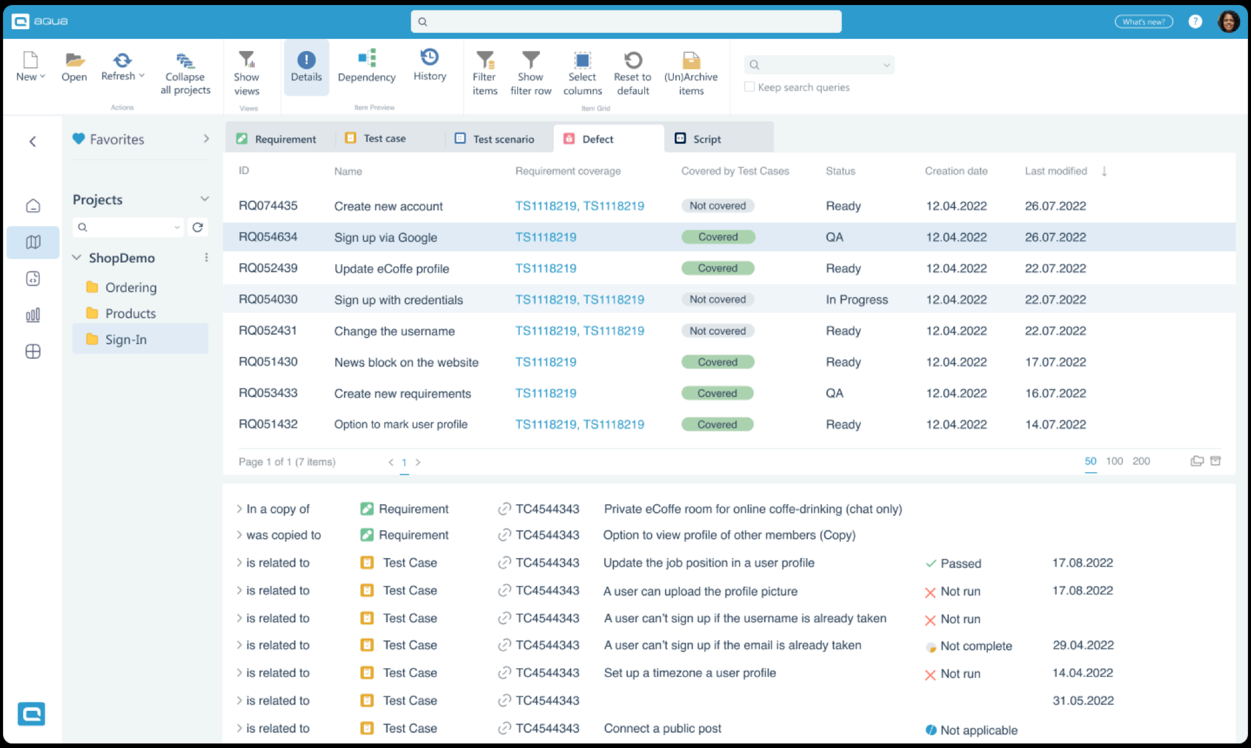
Synthetic Test Data are either manually created or created by data generation tools.

Representative Test Data are usually taken from the production environment and then anonymized.

8.4. Test Suite

Test Suite is a collection of Test Scenarios, Test Cases, or Test Scripts to be executed in a specific test run.

Test Suite often contains detailed instructions or goals for each collection of test items and information on the system configuration to be used during testing.



ID	Name	Requirement coverage	Covered by Test Cases	Status	Creation date	Last modified
RQ074435	Create new account	TS1118219, TS1118219	Not covered	Ready	12.04.2022	26.07.2022
RQ054634	Sign up via Google	TS1118219	Covered	QA	12.04.2022	26.07.2022
RQ052439	Update eCoffe profile	TS1118219	Covered	Ready	12.04.2022	22.07.2022
RQ054030	Sign up with credentials	TS1118219, TS1118219	Not covered	In Progress	12.04.2022	22.07.2022
RQ052431	Change the username	TS1118219, TS1118219	Not covered	Ready	12.04.2022	22.07.2022
RQ051430	News block on the website	TS1118219	Covered	Ready	12.04.2022	17.07.2022
RQ053433	Create new requirements	TS1118219	Covered	QA	12.04.2022	16.07.2022
RQ051432	Option to mark user profile	TS1118219, TS1118219	Covered	Ready	12.04.2022	14.07.2022

Page 1 of 1 (7 items) 1 50 100 200

In a copy of	Requirement	TC4544343	Private eCoffe room for online coffe-drinking (chat only)		
was copied to	Requirement	TC4544343	Option to view profile of other members (Copy)		
is related to	Test Case	TC4544343	Update the job position in a user profile	✓ Passed	17.08.2022
is related to	Test Case	TC4544343	A user can upload the profile picture	✗ Not run	17.08.2022
is related to	Test Case	TC4544343	A user can't sign up if the username is already taken	✗ Not run	
is related to	Test Case	TC4544343	A user can't sign up if the email is already taken	Not complete	29.04.2022
is related to	Test Case	TC4544343	Set up a timezone a user profile	✗ Not run	14.04.2022
is related to	Test Case	TC4544343			31.05.2022
is related to	Test Case	TC4544343	Connect a public post	Not applicable	

It may also contain pre-requisite states or steps, and descriptions of the following tests.

Key Objectives

Test Suite is a container that has a set of tests which helps testers in executing and reporting the test execution status.

In a Test Suite, the Test Cases or Test Scripts are organized in a logical order — for example, the Test Case for registration will precede the Test Case for login.

Test Suite can take one of the following states:

- Active
- In progress
- Completed

Test Suite Types

Test Suites are used to group similar Test Cases together.

Test Suites are often grouped for the following types:

- **Smoke test suite** — Test Cases collection that performs a basic validation for the majority of the product's functional areas and is executed after each product build, before the build is promoted for use by a larger audience
- **Sanity test suite** — Test Cases collection that ensures basic software functionality and serves as the first validation level performed after changes are made to the product
- **Critical Path test suite** — Test Cases collection that crosses the software boundaries and ensures that the integration points between products are exercised and validated
- **Functional Verification test suite** — Test Cases collection that focuses on a specific software function and ensures that several aspects of a specific feature are tested
- **Regression test suite** — Test Cases collection used to perform a regression analysis of the functional software areas, often added into multiple Test Suites and Test Plans

8.5. Test Plan

Test Plan is a product-level document that describes the test objectives, schedule, estimation, deliverables, and resources required for the software testing.

Key Objectives

Test Plan serves a template to conduct software testing activities as a defined process monitored and controlled by the Test manager.

Test Plan must be consistent with the company's Test Policy and Test Strategy documents.

Key Components

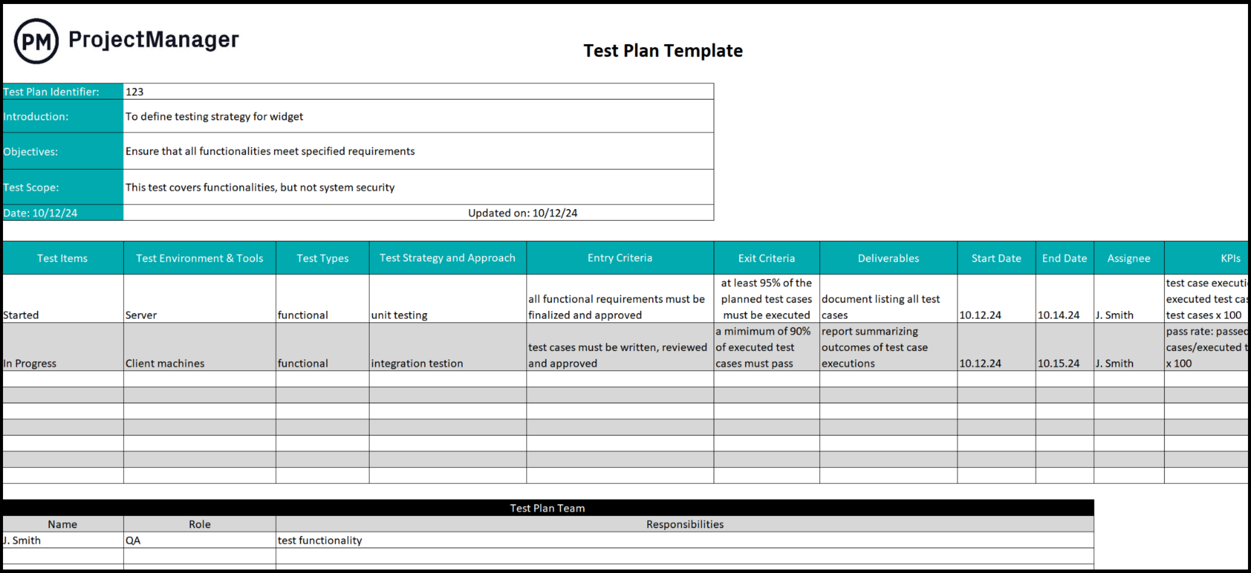
Test Plan should usually include:

- Test Plan identifier
- References
- Introduction
- Test items
- Software risk issues
- Features to be tested
- Features not to be tested
- Testing approach
- Test Pass/Fail criteria
- Suspension criteria and Resumption requirements
- Test deliverables
- Test environment setup
- Staffing and training needs
- Team member responsibilities
- Testing schedule
- Risks and contingencies planning
- Approvals

- Glossary

— and other chapters.

Test Plan components mentioned above are not mandatory and may be omitted on necessity.



ProjectManager

Test Plan Template

Test Plan Identifier:	123
Introduction:	To define testing strategy for widget
Objectives:	Ensure that all functionalities meet specified requirements
Test Scope:	This test covers functionalities, but not system security
Date: 10/12/24	Updated on: 10/12/24

Test Items	Test Environment & Tools	Test Types	Test Strategy and Approach	Entry Criteria	Exit Criteria	Deliverables	Start Date	End Date	Assignee	KPIs
Started	Server	functional	unit testing	all functional requirements must be finalized and approved	at least 95% of the planned test cases must be executed	document listing all test cases	10.12.24	10.14.24	J. Smith	test case executi executed test cas test cases x 100
In Progress	Client machines	functional	integration testion	test cases must be written, reviewed and approved	a mimimum of 90% of executed test cases must pass	report summarizing outcomes of test case executions	10.12.24	10.15.24	J. Smith	pass rate: passed cases/executed t x 100

Test Plan Team

Name	Role	Responsibilities
J. Smith	QA	test functionality

Often, companies create their own formats which are roughly based on any standard.

Benefits

Test Plan document has multiple advantages as it:

- Helps people outside the QA team — developers, business managers, customers — understand the details of testing
- Guides QA team's thinking towards test procedures to be completed
- Documents key aspects — test estimation, test scope, test strategy — for future review and reuse in other projects.

8.6. Test Strategy

A **Test Strategy** is a high-level, organization-wide document that outlines the approach, objectives, and execution plan to test a product.

It serves as a blueprint to ensure testing aligns with business goals, technical requirements, and quality standards.

Key Objectives

The purpose of the Test Strategy is to:

- Define the scope, focus, and methodologies of testing
- Align testing efforts with project timelines and business goals
- Ensure consistency across teams
- Optimize resource allocation — tools, budget, personnel
- Mitigate risks through structured planning

Key Components

The main chapters of the Test strategy are as follows:

- Scope and Objectives — What will be tested and why
- Testing Types — Functional, non-functional, regression, etc
- Test Levels — Unit, integration, system, UAT, etc
- Test Environment — Hardware, software, and tools required
- Test Data Management — How test data will be created, stored, and anonymized
- Roles and Responsibilities — Who designs, executes, and approves tests
- Risk Analysis — Identify high-risk areas and mitigation plans
- Entry and Exit Criteria — Conditions to start and stop testing
- Automation Approach — Tools, coverage, and CI/CD integration
- Defect Management — How bugs are logged, prioritized, and retested

- Metrics and Reporting — Key performance indicators, KPIs, and reporting frequency
- Deliverables — Test Plans, Test Cases, Test Result Reports, and Audit Logs

Test Strategy Role

A well-defined Test Strategy ensures testing is structured, efficient, and goal-driven.

It bridges the gap between business objectives and technical execution, reducing costs while maximizing quality.

8.7. Test Policy

Test Policy is the company-level document which defines the test principles adopted by an organization.

Test policy is determined by the company's Chief Executive Officers — CEOs — which provide an organizational insight for the test activities.

Key Objectives

The Test Policy commonly describes:

- The place of testing in the company
- Test objectives of the organization
- Testing process definition
- Test effectiveness measurement
- Test processes improvement approach

8.8. Test Management Systems

Test Management System — TMS — is an application designated to help teams manage their software testing processes effectively.

TMS can cover all test documentation levels and provide a centralized location for storing and managing Test Cases, Scenarios, Requirements, and Defects.

Key Objectives

Test Management Systems play a crucial role in ensuring software quality by automating the following processes:

- Requirements Traceability
- Project Tasks Tracking
- Test Case Management
- Automated Scripts Execution
- Defect Tracking
- Reporting and Metrics Provision

Benefits

The main advantages of Test Management Systems are as follows:

- Testing process consolidation
- Efficient data access and analysis
- Effective communication across teams

IX. Software Quality Defects

9. Software Quality Defects

Software Quality Defect — also called **Bug** or **Fault** — is a flaw, error, or imperfection in a software product that causes it to behave in unintended ways or fail to meet specified requirements.

Defects represent deviations between actual and expected requirements, specifications, or user expectations, resulting in either functional flaws — crashes, wrong outputs, etc — or non-functional shortcomings — slow performance, security vulnerabilities, and so on.

Key Characteristics

Defects may be classified using the following key characteristics:

1. **Root Causes:**
    - Coding errors
    - Flawed design or architecture
    - Misunderstood requirements
    - Environmental incompatibilities
2. **Impact Levels:**
    - Critical — System crashes, data loss
    - Major — Core features malfunction
    - Minor— Cosmetic issues
3. **Detection Methods:**
    - Testing
    - Code reviews
    - User feedback

Defect Management Role

Defect management plays a crucial role for the following main reasons:

- **Cost** — fixing defects post-release is many times costlier than during development
- **Reputation** — usually, one in four users abandon the software after the app crashes
- **Compliance** — defects in healthcare or fintech software can violate regulations and bring to penalties

Best Practices

The best practices to reduce software defects should include:

- **Shift-Left Testing** — catching bugs in advance introducing unit or integration tests as early as possible
- **Static Analysis** — dedicated tools implementation for the code quality checks
- **Peer Reviews** — 60–90% of defects avoidance via the regular code reviews
- **Automated Regression Testing** — preventing the reintroduction of old bugs by automating the test cases for known issues

---

9.1. Defect Classification

The software quality defects are commonly classified using the following criteria.

Defect Classification By Severity

Considering the impact on a system, software defects may be treated as:

- **Critical** — Causes system crash, data loss, or complete failure:
  - Database corruption
  - Application crash on startup etc
- **High** — Major functionality broken but system remains operable:
  - Login failure
  - Payment processing error and so on
- **Medium** — Partial functionality loss with workarounds available:
  - Search returns incomplete results
  - UI formatting issues and others
- **Low** — Cosmetic issues with no functional impact:
  - Spelling errors
  - Minor alignment problems and alike

Defect Classification By Priority

By urgency of fix, software defects may be divided into the following groups:

- **Immediate** — Must be fixed immediately, blocking further work and should be fixed the next build

- **High** — Important to fix soon, affects key functionality and should be fixed in current release
- **Medium** — Has workarounds and should be fixed the next release
- **Low** — Minor issues which may be fixed when convenient, as a rule in a future release

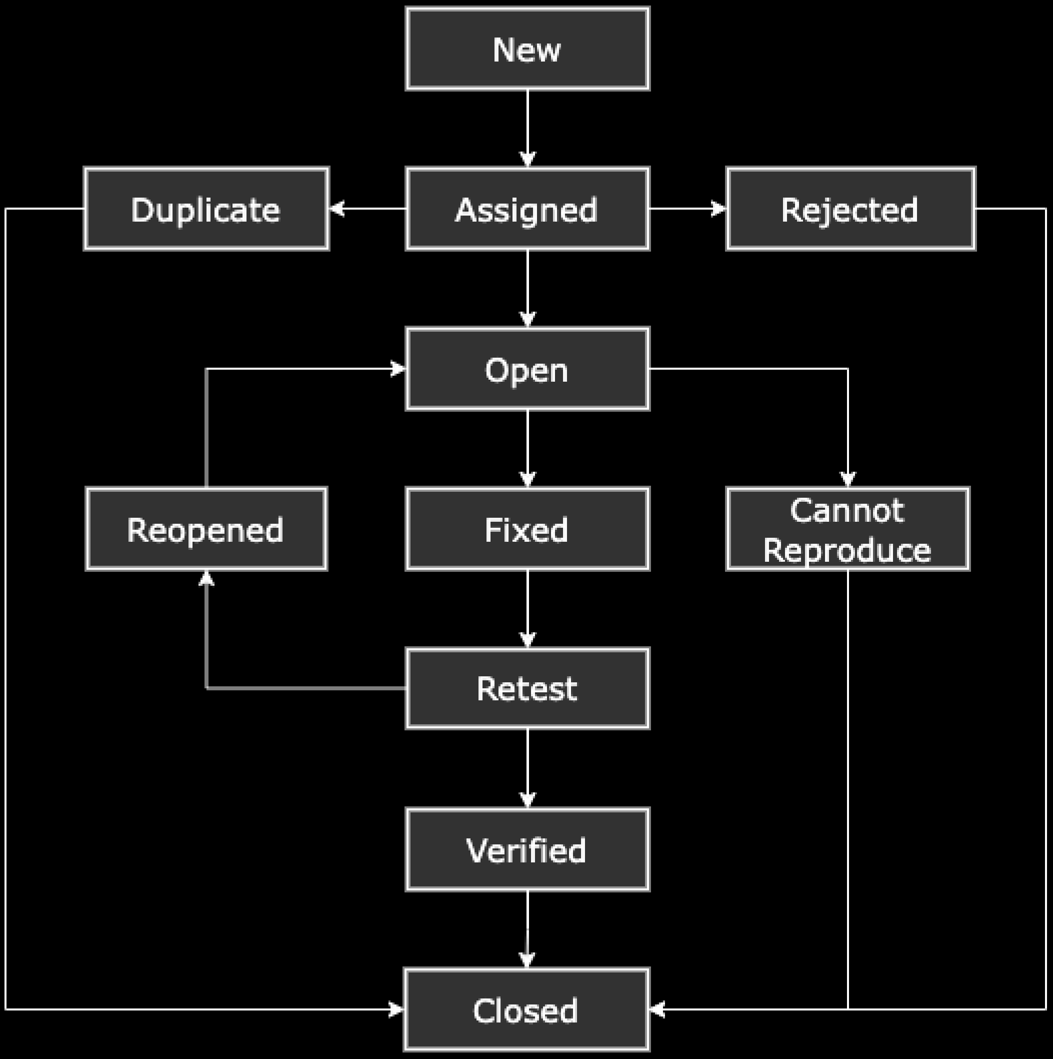
Defect Classification By Origin

Considering the source of a defect, they may be split into the following categories:

- **Requirements Defects** — Incorrect or missing requirements:
  - Unclear business rules
  - Conflicting requirements etc
- **Design Defects** — Architecture or design flaws:
  - Poor database design
  - Insecure architecture etc
- **Coding Defects** — Implementation errors:
  - Syntax errors
  - Logic errors
  - Boundary condition issues etc
- **Testing Defects** — Errors in test cases or environment:
  - Incorrect test data
  - Environment configuration issues etc
- **Integration Defects** — Component interaction issues:
  - API compatibility problems
  - Data format mismatches and others

9.2. Defect Life Cycle

The Defect Life Cycle may be represented with the following scheme.



Defect Life Cycle

Detailed State Transitions

Understanding software defects and their life cycle is fundamental to effective quality assurance.

A well-managed defect process includes as a rule following Detailed State Transitions:

- New — Defect is identified and logged for the first time
- Assigned — Defect is assigned to developer
- Duplicate — Defect already reported by someone else
- Rejected — Defect is invalid or not a bug
- Open — Developer accepts and starts working on the defect
- Fixed — Developer has implemented the fix
- Retest — Tester verifies the fix
- Reopened — Fix is incomplete or defect reappears
- Verified — Fix is confirmed to be working correctly
- Closed — Defect is formally closed in the system

9.3. Defect Report

Defect Report — also called **Bug Report** — is a formal document that describes a software flaw or failure that causes it to behave unexpectedly or produce incorrect results.

It serves as the primary communication tool between testers, developers, and stakeholders to track and resolve software issues.

Key Components

The essential Bug Report components are as follows:

- Defect ID — Unique identifier
- Summary — Brief, descriptive title
- Reported By — Who found the defect
- Report Date — When defect was found
- Component — Affected module of application
- Severity — Impact on system functionality
- Steps to Reproduce — Exact sequence to recreate defect
- Expected Result — What should happen according to SRS
- Actual Result — What actually happens
- Environment — Where defect was found
- Evidence — Screenshots, records, logs, database snapshots

Key Characteristics

Effective Defect Reports are:

- Clear — Easy to understand without ambiguity
- Concise — No unnecessary information
- Complete — Contains all needed information
- Consistent — Follows organizational standards
- Reproducible — Others can recreate the issue

Writing effective Defect Reports requires:

- Using objective language — *"Clicking 'Submit' button displays error message: 'Database connection failed'"*, not *"The system is broken when you try to save data"*
- Being specific and detailed — *"Dropdown shows 5 items instead of expected 7: missing 'Admin' and 'Manager' roles"*, not *"Dropdown options are wrong"*
- Placing one defect per report — *Defect A: "Login button not working", Defect B: "Password field allows invalid characters"*, not *"Login and password fields have issues"*
- Including visual evidence:
    - Using arrows or circles to highlight issues in screenshots
    - Recording videos for complex multi-step defects
    - Capturing relevant log entries with timestamps

A well-written Defect Report is more than just a bug description — it's a critical communication tool that drives quality improvement.

Key Objectives

Effective defect reporting:

- Accelerates problem resolution through clear communication
- Provides data for quality metrics and process improvement
- Ensures accountability through proper tracking
- Supports informed release decisions

The quality of Defect Reports directly impacts the efficiency of the development process and the quality of the final product.

Investing time in writing clear, complete, and professional Defect Reports pays significant dividends throughout the SDLC.

X. Software Testing Classification

10. Software Testing Classification

Software Testing Classification is a systematic categorization of the software testing categories according to their purpose criteria.

The perception of classification allows making informed decisions and contributing to delivering reliable software products.

Key Objectives

The goals of the Software Testing Classification include:

- Comprehensive Segmentation
- Testing Strategy Tailoring
- Efficient Resource Allocation
- Forcible Risk Mitigation
- Responsive Quality Assurance

The choice of the testing type depends on such factors like:

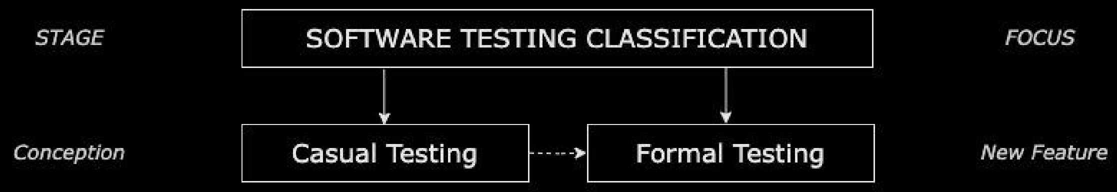
- System scope
- SDLC stage
- Project budget
- Software significance

10.1. Software Conception Stage Categories

Software Conception is the stage to define the very approach for the documentation strategy of the upcoming development, testing, and related processes.

There are two major testing types crucial for the Software Conception stage:

- Casual testing
- Formal testing



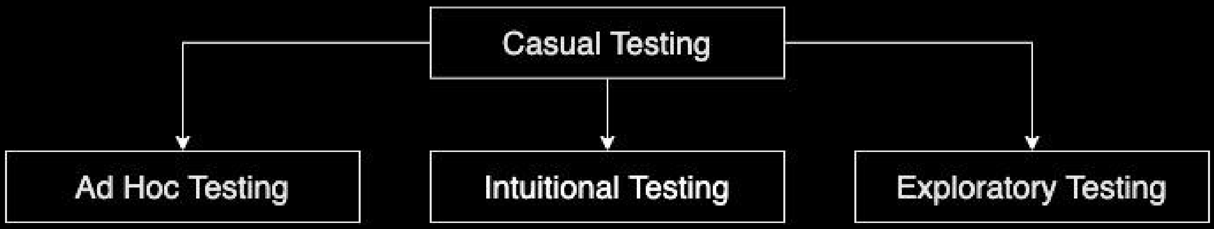
The dashed line illustrates the transition from Casual to Formal Testing, because Formal Testing provides the reliability, repeatability, and accountability that Casual Testing inherently lacks.

Casual Testing

Casual testing is a type of Software testing performed without any dedicated planning, documentation, and procedures.

There are three major types of Casual testing:

- Ad hoc testing
- Intuitional testing
- Exploratory testing



Ad Hoc Testing

Ad Hoc Testing — from latin "To this *purpose*" — is the least formal type of Casual testing, also called **Random Testing**.

Ad Hoc tests are only run once, unless a bug is discovered, and since the very testing is not documented, the found defects are difficult to reproduce.

The strength of Ad Hoc Testing is in its applicability during the earliest stages of development life cycle so the evident bugs are found quickly.

	Test Planning	Documentation	Prior Experience
Ad Hoc Testing	No	No	No
Intuitional Testing	No	No	Yes
Exploratory Testing	No	Yes	Yes

Intuitional Testing

Intuitional Testing — often called **Error Guessing** — is a type of Casual Testing based on prior testing experience and expertise.

Error Guessing has no explicit testing rules and depends on the situation entirely.

It is solely determined by the past experience and intuition of a Quality Assurance Engineer who may possibly know the problem areas which commonly invoke the software failures.

Exploratory Testing

Exploratory Testing is a type of Casual Testing performed as a process of simultaneous learning, test design, and test execution.

Cem Kaner, who coined the term in 1984, defines exploratory testing the following way:

*"**Exploratory testing** is a style of software testing that emphasizes the personal freedom and responsibility of the individual tester to continually optimize the quality of his/her work by treating test-related learning, test design, test execution, and test result interpretation as mutually supportive activities that run in parallel throughout the project."*

While the software is being tested, the tester learns things that together with experience and creativity generates new tests to run.

Checklist is the only artifact used to structurize the Exploratory Testing in order not to waste time repeating the same tests.

Formal Testing

Formal Testing is a type of software testing performed with proper planning, documentation, and procedures.

It is carried out with meticulous Test Cases documentation and systematically adheres to the Software testing life cycle.

Formal Testing is more expensive due to the outlays for the test case preparation, personnel training, and documentation authoring.

Origins

Formal Testing has three primary documentation origins:

- User Stories
- Business Scenarios
- Software Requirements Specification

User Stories

User Stories are concise statements used in software development which communicate a single feature, task, or goal from the User's perspective.

A typical User Story format is: "As a Type of User, I want Some Goal so that Some Reason", for instance:

"As an online shopper, I want to be able to filter products by category so that I can easily find the items I'm interested in."

Business Scenarios

Business Scenarios are loosely defined descriptions of specific situations explaining the User's view on achieving a goal or completing a task.

They reveal the Customer's perspective on a software feature omitting the process of its implementation and can be depicted as a series of steps.

Business Scenarios provide context and comprehension on how a particular type of person might interact with a product or service.

Software Requirements Specification

Software Requirements Specification is the documented set of expectations which encompasses all functional aspects to form a test execution basis for multiple features, constraints, and capabilities.

It serves as a single source of truth both for the program code and Test Case development during the next stages of the Software development life cycle.

Formal Testing Types

Respectively, there are three major types of Formal testing:

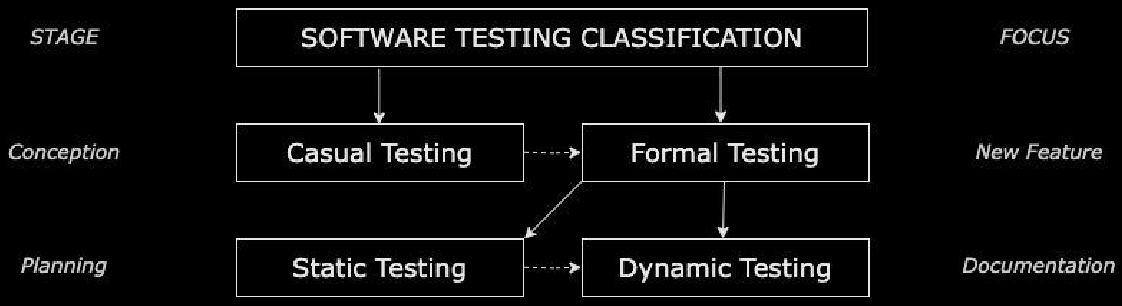
- Story-based testing
- Scenario-based testing
- Specification-based testing

10.2. Software Planning Stage Categories

Software Planning is the stage to determine the very testing approach for the following software development life cycle.

There are two major testing types crucial for the Planning stage:

- Static Testing
- Dynamic Testing



The dashed line indicates a trend of favoring Dynamic Testing over Static Testing, as the former excels at detecting the wider range of specific defects, while the latter is only superior for early defect prevention.

Static Testing

Static Testing is a type of Formal Testing performed without the software code execution.

For this reason, Static Testing is also called:

- Documentation testing
- Non-execution testing

- Verification testing
- Testing review

Tested Documents

The most important statically tested documents include:

- Requirements Analysis documents:
    - User Stories
    - Business Scenarios
    - Software Requirement Specifications
- Software Design Specifications
- Test design documentation:
    - Test Plans
    - Test Cases
- Test development documentation:
    - Source code
    - Test Scripts
- Release documentation:
    - User manuals
    - Helps

Static Testing Purpose

Static Testing, requiring no software run, allows catching crucial bugs and ambiguities on the earliest stages of the development.

The earlier logic mistakes and requirement contradictions are found, the lesser the cost of their correction and the straighter the process of software development.

Dynamic Testing

Dynamic Testing is a type of Formal Testing performed with the software code execution.

It allows validating the software functional behavior, memory, processes, and overall performance in dynamics.

That's why Dynamic Testing is also called **Execution Testing** or **Validation Testing**.

Dynamic Testing Purpose

Dynamic Testing focuses on evaluating the runtime behavior of the software code and observing its behavior under various conditions.

It aims to ensure that the software functions correctly during and after installation, and implies actual output collation with the expected one.

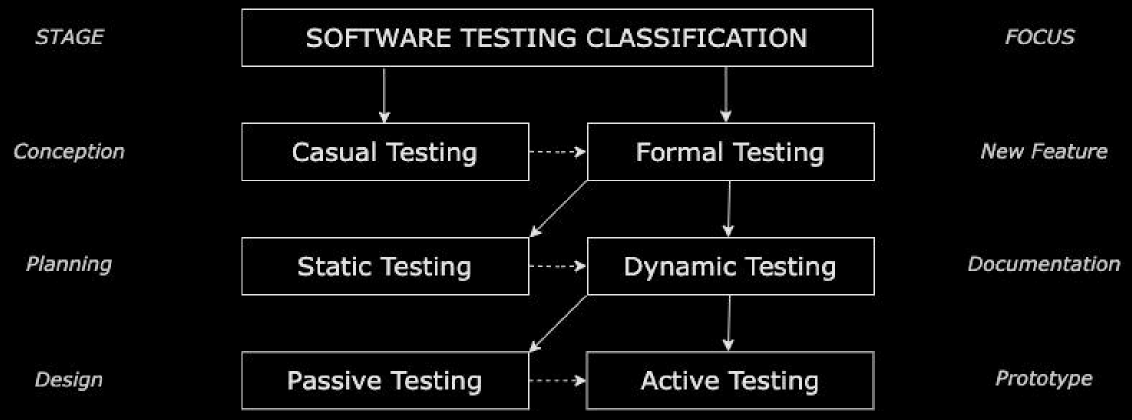
Dynamic Testing may begin before the software is ready; certain units or components may be tested dynamically using stubs, drivers, or debuggers.

10.3. Software Design Stage Categories

Software Design is the stage which allows acquiring the first operable Prototype or Minimum Viable Product — MVP.

There are two major software testing types during the Design stage:

- Passive Testing
- Active Testing



The dashed line represents the trend of transitioning from Passive to Active Testing driven by the need for speed, reliability, and continuous feedback in modern software development.

Passive Testing

Passive testing is a type of Dynamic Testing performed through the code execution with no User interactions with the software.

Passive Testing means verifying the system behavior through the offline runtime verification, log analysis, and stack trace inspection.

Active Testing

Active Testing is a type of Dynamic Testing performed through the code execution along with User interactions with the software.

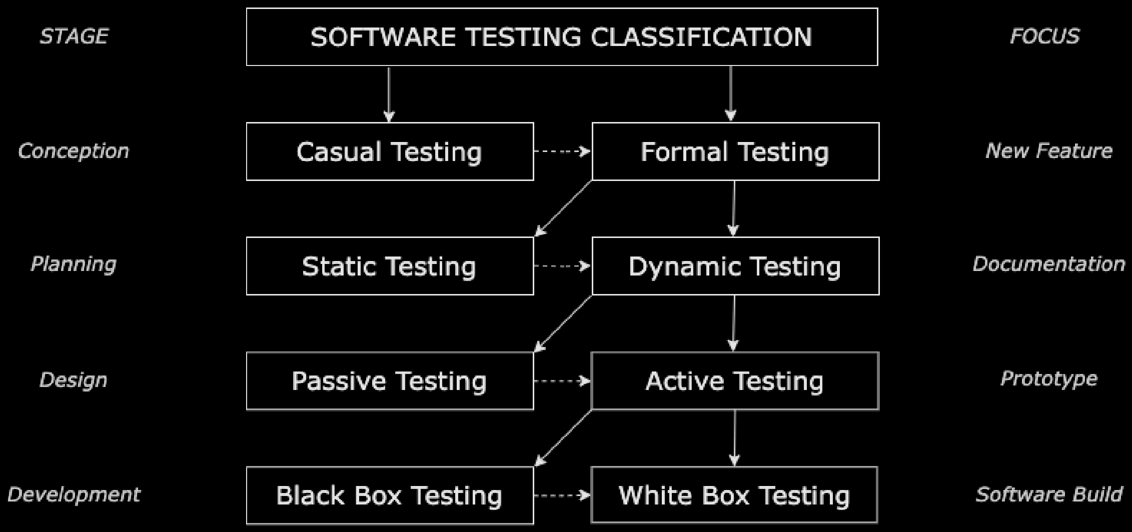
Active Testing is dedicated to evaluate the robustness, reliability, and optimal performance of the software in dynamic environments.

10.4. Software Development Stage Categories

Software Development is the stage which provides both executable and testable builds of a target software.

There are three testing types performed during the Development stage:

- Black Box Testing
- Gray Box Testing
- White Box Testing



The dashed line depicts the trend of transitioning from Black Box Testing to White Box Testing as defect prevention is more efficient and cost-effective than detection.

Black Box Testing

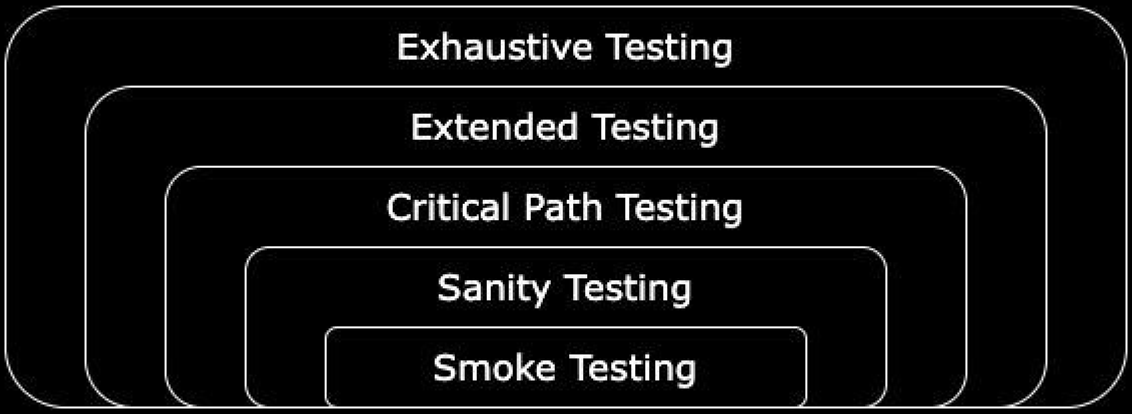
Black Box Testing is a type of Active Testing performed with no knowledge of software code, internal structure, and implementation details.

The term symbolizes the inability to see the inner software implementation so that only testers' experience can be applied.

Black Box Testing requires manual quality assurance skills which can better imitate the external behavior of terminal users.

There are five Black Box Testing levels, depending on the tested scope:

- Smoke Testing
- Sanity Testing
- Critical Path Testing
- Extended Testing
- Exhaustive Testing



Smoke Testing is the minor level of Black Box Testing performed to reveal evident bugs severe enough to reject a software build promoted for further verification.

Sanity Testing is a type of Black Box Testing performed to quickly evaluate the software operability against the basic set of verifications.

Critical Path Testing — also called **End-to-End testing** — is a type of Black Box Testing carried out to analyze the functionality most commonly employed by the majority of software users.

Extended Testing is a type of Black Box Testing performed to explore all declared functionality specified in the Software requirement specification.

Exhaustive Testing — or **Complete testing** — is a type of Black Box Testing performed to confirm that software functions correctly under every possible situation.

While Exhaustive Testing is not actually achievable, diligent testing helps create robust applications with minimal defects.

Gray Box Testing

Gray Box Testing is a type of Active Testing performed with a partial notion of software code, internal structure, and implementation details.

The term implies that the approach is a combination of both Black Box and White Box Testing.

White Box Testing

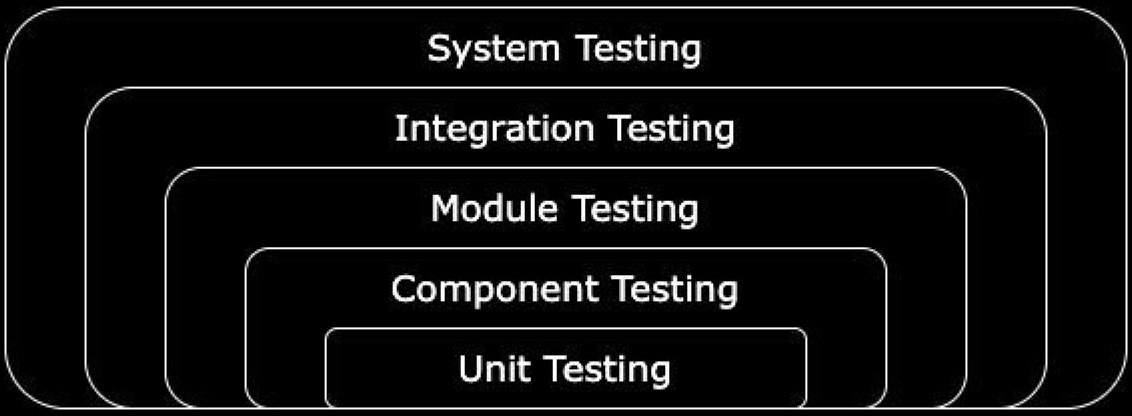
White Box Testing is a type of Active Testing performed with the knowledge of software code, internal structure, and implementation details.

The term refers to the see-through box concept which symbolizes the ability to see through the software's outer shell into its inner state.

White Box Testing requires software or test developers to create Test Scripts making this type of testing more expensive.

There are five White Box Testing levels, depending on the tested scope:

- Unit Testing
- Component Testing
- Module Testing
- Integration Testing
- System Testing



Unit Testing is the minor level of White Box Testing, typically performed by developers, which spots on testing the individual units, in isolation.

Component Testing is a type of White Box Testing performed on a compound feature derived from the independent units coalition, in isolation.

Module Testing is a type of White Box Testing which focuses on testing the independent Modules made of grouped Components, in isolation.

Integration Testing is a type of White Box Testing performed on a set of modules joined in a Sub-system entity to verify the combined operability.

System Testing is a type of White Box Testing conducted on the integrated software to evaluate its compliance with the requirements specified.

10.5. Software Testing Stage Categories

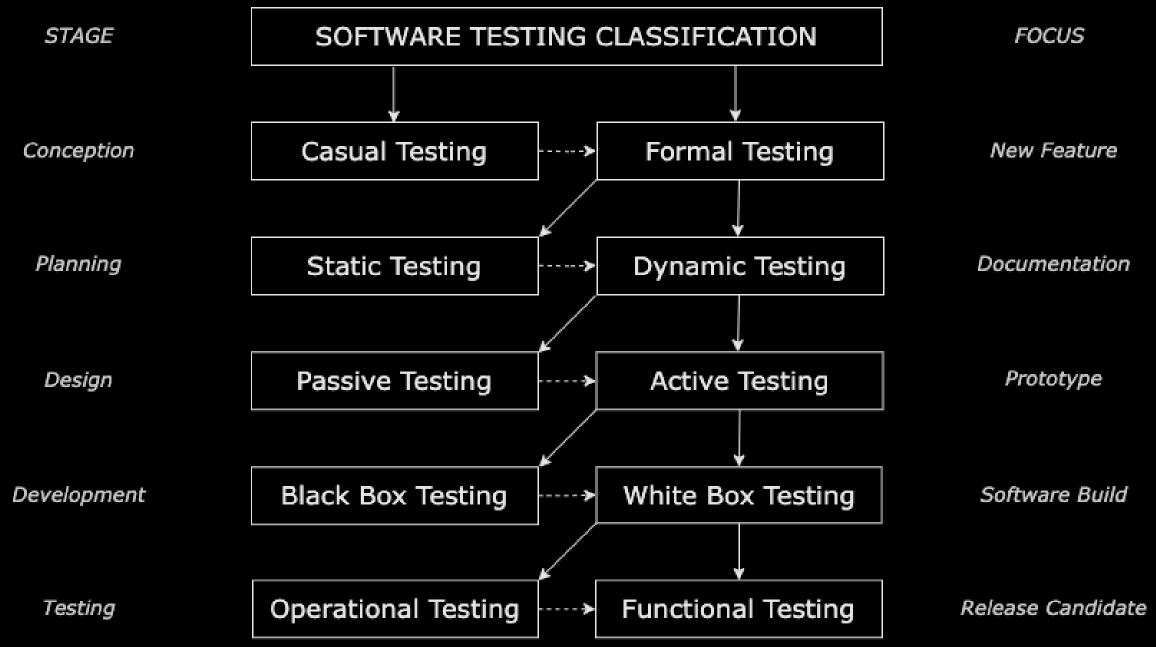
Software Testing is the stage of repetitive software build validations to fetch it to the state of a Release Candidate.

Release Candidate — RC — is a potentially shippable version of software that is feature-complete and considered ready for final testing before official release.

It represents the final stage of the development cycle where the software is deemed likely to become the production version unless critical issues are discovered.

There are two types of testing performed during the Software Testing stage:

- Operational Testing
- Functional Testing



The dashed line reveals the greater importance of Functional Testing over Operational Testing underscoring that working software is a prerequisite for evaluating how well it works in terms of speed, security, or reliability.

Operational Testing

Operational Testing is a type of White Box Testing performed to validate operational — non-functional — aspects of the software.

Non-functional aspects are those that reflect the quality of the product, particularly in the context of the suitability perspective of its users, and are never related to a specific software function:

- Performance
- Usability
- Reliability
- Scalability
- Security etc

Operational Testing is designed to verify the software readiness of the non-functional aspects which are never addressed by functional testing.

It validates the way software operates, as opposed to the functional behaviour verification.

There are four major types of Operational Testing:

- Installation Testing
- Usability Testing
- Payload Testing
- Security Testing

Operational Testing Purposes

Operational Testing refers to the software aspects that may not be related to a particular function or User action.

Operational Testing implies the software requirements testing that are not functional in nature but important enough for the system operability.

Operational requirements reflect the software quality as a whole, in the context of its users suitability perspective, particularly.

Functional Testing

Functional Testing is a type of White Box Testing performed to validate the software against its functional requirements and specifications.

Functional Testing Purposes

Functional Testing serves to verify the software actions by providing an input and estimating the output against the documented conditions.

A software function has two distinctive non-operational features crucial for the Functional Testing:

- Evaluability — Expected and actual results' existence
- Bilaterality — Positive and negative testing approaches

Evaluability

Software function evaluability refers to the simplicity and effectiveness with which this function can be assessed, measured, and judged against specific criteria.

Bilaterality

Bilaterality is a distinctive property of a software function which is achieved due to its reciprocity for the testing purposes.

Bilaterality allows to split Functional Testing into two major types:

- Positive Testing
- Negative Testing

Positive Testing

Positive Testing — or **Happy Path Testing** — is a type of Functional Testing performed on a software by providing valid, proper, and correct data as input.

Positive Testing verifies whether the software behaves as expected with positive and awaited user inputs, or not, to confirm the software viability.

Positive Testing should precede the Negative one because defects found through the Positive tests prevent the necessity for further testing.

Negative Testing

Negative Testing is a type of Functional Testing performed on a software by providing invalid, corrupt, or improper data as input.

Negative testing verifies whether the software behaves as expected with negative or unwanted user inputs, or not, to uncover issues in error handling.

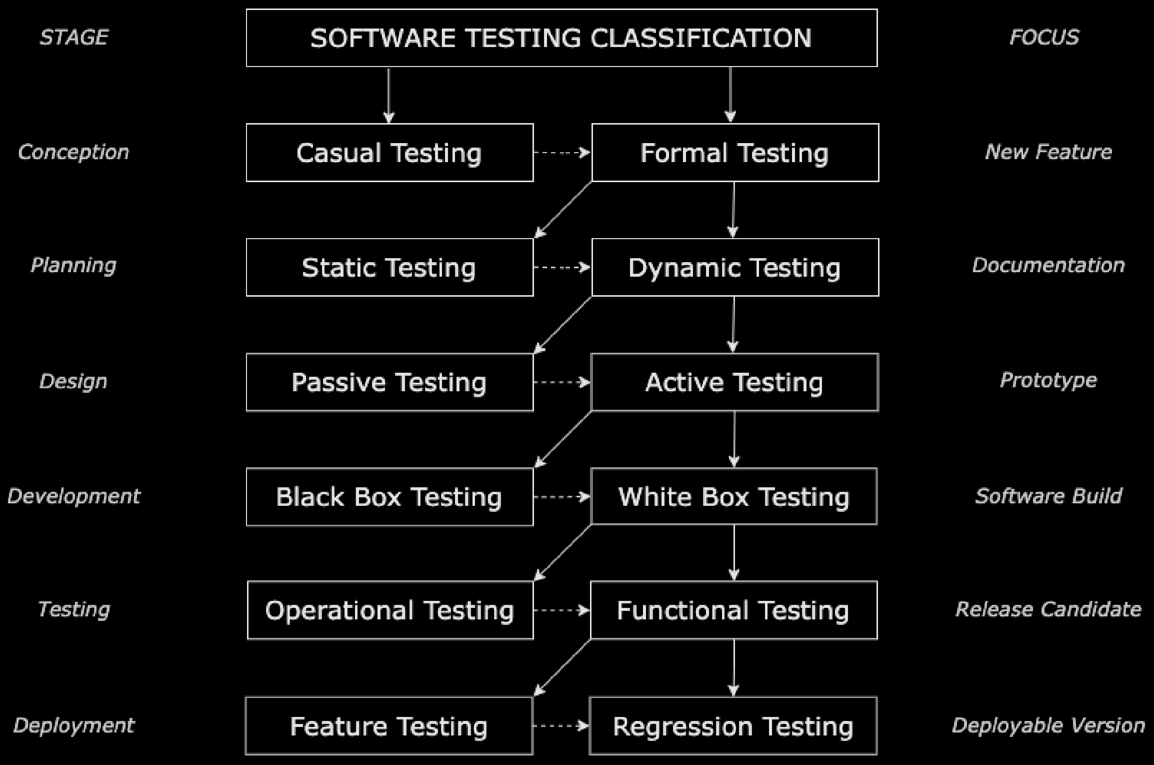
It means, a software might and actually should work correctly under improper contexts, that is under Negative Testing.

10.6. Software Deployment Stage Categories

Software Deployment is the stage where the Release Candidate is being tested, approved, and rendered to the Client.

There are two testing types performed during the Deployment stage:

- Feature Testing
- Regression Testing



The dashed line reflects the common and critical mindset in the professional software industry that protecting existing functionality is a higher priority than validating the new one.

Feature Testing

Feature Testing — or **New Feature Testing** — is a type of Functional Testing performed to verify the new code or environment amendments comply with the software requirements.

The goal of new Feature Testing is to ensure that augmented functions perform as expected, introduce no defects, and maintain overall software quality.

Regression Testing

Regression Testing is a type of Functional Testing performed to verify the new code or environment amendments retain the previously achieved quality.

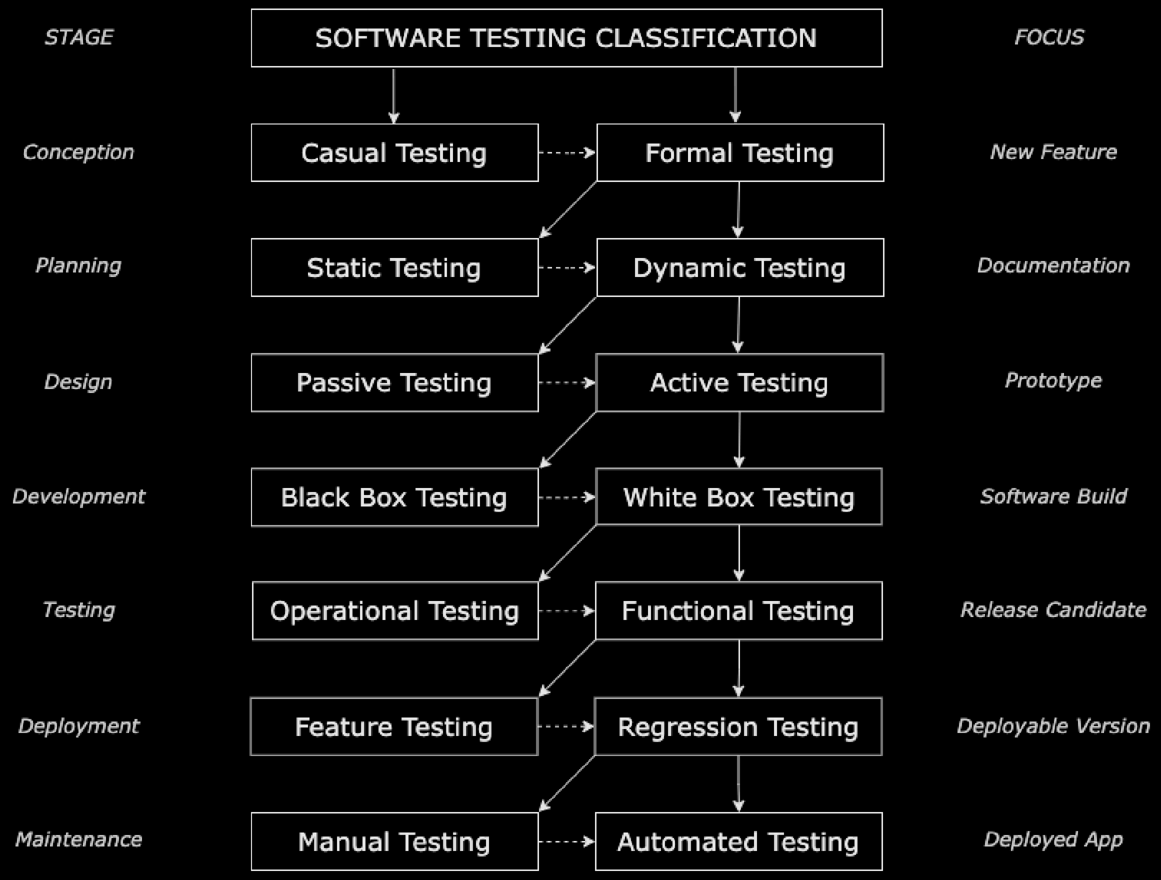
Regression Testing implies that already developed tests are re-executed to verify the influence of changes on the functionality that existed before.

10.7. Software Maintenance Stage Categories

Software Maintenance is the stage to enhance the software test coverage to assure its higher quality prior to the next SDLC turn.

There are two types of Regression Testing performed during the Maintenance stage:

- Manual Testing
- Automated Testing



The dashed line from Manual Testing to Automated Testing emphasizes the greater importance of the last one with core reason that Automated Testing enables speed, scale, and reliability in a way that Manual Testing cannot, making it the backbone of any team that needs to release software frequently and reliably.

Manual Testing

Manual Testing is a type of Regression Testing executed manually.

Manual Testing means all the essential software feature verifications and test report generation are performed without the automated testing tools.

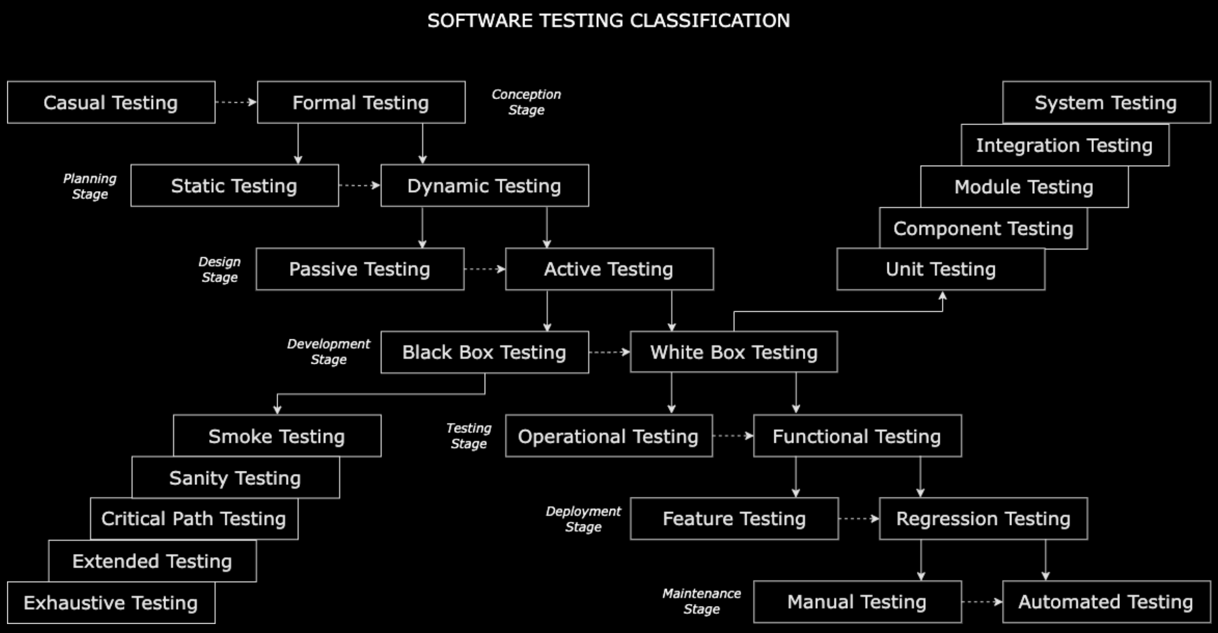
Automated Testing

Automated Testing is a type of Regression Testing executed automatically by running the test scripts with the automated testing tools applied.

Automated Testing supposes the use of appropriate automation tools to develop, enhance, and execute the test scripts to validate the software.

10.8 Software Testing Classification Scheme

Software Testing Classification may finally be depicted by the following scheme.



According to the scheme, Software Development is the most time and effort consuming, resource extensive, and responsible stage considering the goals of the Software testing life cycle.

Testing is crucial for the Software Development stage as it bridges development and real-world usage, ensuring software is reliable, secure, and user-ready.

Without it, even perfectly designed systems risk catastrophic failures.

XI. Practice

11. Practice

Learning Software quality assurance fundamentals and then moving to a real-life project is one of the most effective ways to build a valuable and rewarding career.

Let's break down why — and create a practical path forward.

Why

The software industry has realized that you cannot test quality at the end — it must be built in throughout the process.

This has created a massive demand for skilled QA professionals who understand both theory and practice — every app, website, and digital system needs to be tested.

Whether you stay in QA or move into development, management, or product design, understanding QA fundamentals makes you more thorough, user-focused, and valuable — you learn to think about edge cases, risks, and the user experience in a way that pure developers often overlook.

QA theory gives you a framework of what to test, why to test it, and practice shows you the reality of how things actually break, how to communicate with developers, how to prioritize — and this combination is powerful!

Finding a critical bug before it reaches users is incredibly satisfying — you directly prevent frustration, protect company revenue, and safeguard the user experience.

You transition from passive learning to actively creating value.

How

Don't jump into a massive project immediately — follow this progression to build confidence and skills effectively.

- Step 1. Solidify the Fundamentals — ensure you have a grip on core concepts:
    - SDLC & STLC
    - Test Types
    - Basic Test Documentation
    - Other Core Notions
- Step 2: Practice on a Controlled Real Project — test something that exists but where the stakes are low:
    - Choose a popular website or open-source app — anyone you love most
    - Create a simple Checklist — try to add the most valuable checks there, don't chase for the quantity
    - Write formal Test Cases — don't just click around, write detailed cases using a dedicated tool
    - Execute Tests and Log Bugs — find actual flaws, UI glitches, or usability issues and report them in a bug-tracker
- Step 3: Join a Real-Life Project:
    - Open Source Software
    - Volunteer Projects
    - Internships or Junior QA Roles

This gives you a portfolio piece and practical experience with the workflow.

The Goal is — You're ready for a team environment.

When

This disordered reality is where the real learning happens.

Your theoretical knowledge is the map — practicing on a real project is the journey where you learn to navigate the actual terrain.

You will get stuck, you will be confused, but you will learn exponentially faster than through theory alone.

You need to develop soft skills like communication, persuasion, and patience that are just as important as your technical testing skills.

Stop hesitating, start breaking things.

First, join the t.me/sqafun group in Telegram to connect with other readers of this book, just like you — by joining forces, you will move faster.

Next, if you need my closer standing by, drop me a line in direct messages and I'll try to help you out.

My email al.vorvul@gmail.com is also at your constant disposal for your questions, criticism, and proposals.

The transition from learning SQA to practicing it is not a leap.

It's a series of small, manageable steps.

The best time to start was yesterday.

The second-best time is now.

Go for it.

The tech industry needs more skilled, passionate QA professionals.

Alexander Vorvul

SOFTWARE QUALITY ASSURANCE FUNDAMENTALS

Electronic version

Wydawnictwo Gutenberg Publisher
Łobzowska 15-15
31-139 Kraków

office@gutenbergpublisher.eu

ISBN 978-83-68016-45-1

9 788368 016451