QUASAR #172

INFORMING AND ADVANCING OUR MEMBERS

www.therqa.com

AUGUST 2025 | ISSN 1463-1768

SAFEGUARDING GOOD LABORATORY PRACTICE IN THE DIGITAL AGE: IT SECURITY CONSIDERATIONS FOR GLP TEST FACILITIES

40 PAGES OF ARTICLES

SILENT SHADOWS: UNSPOKEN INCIDENTS IN CLINICAL TRIALS

RETHINKING RECALL MANAGEMENT IN LIFE SCIENCES: FROM REACTIVE TO READY

THE LATEST COURSES P51

SOFTWARE TESTING: MEASURING VENDOR SOFTWARE QUALITY – PART THREE: SOFTWARE QUALITY ANALYTICS

RISK-PROPORTIONATE APPROACH IN ICH GCP E6 (R3): ENHANCING EFFICIENCY AND COMPLIANCE IN CLINICAL TRIALS

RQA

Barry McManus

Hugh O'Neill

SOFTWARE TESTING: MEASURING VENDOR SOFTWARE QUALITY – PART THREE:

SOFTWARE QUALITY ANALYTICS

This series on assessing a vendor's software product quality has discussed the problem of a vendor's QMS not reflecting their software production line processes, where their 'compliant' QMS documentation quality may not correlate with the resulting software product quality. This article is examining how analytics can be leveraged to assess the software product quality. Quasar #170 and #171 discussed how the purpose of verification (testing) is to provide information.

(Note: the following acts as a guide for consideration during vendor discussions as the scope will be determined by the context of the computerised system to be automated and the quality maturity of the vendor).

INTRODUCTION

Software product quality analytics focuses on measuring process outcomes to identify quality trends. These analytics help predict the likelihood of future defects and support decisions about when to continue or conclude testing. Compared to Quality Management Systems (QMS) that operate on infrequent review cycles (e.g. SOP review every two years), Software Quality Analytics enable continuous, real-time improvement of the software production process. It plays a key role in adjusting practices to prevent, reduce or eliminate recurring defects.

During vendor audits, the auditee will describe their QMS, that it is managed by a CAPA process, and emphasise the use of templates. Compliance is demonstrated from documentation conforming to 'a' documented QMS, right? But when the question transitions from 'describe the QMS?' to 'how good is the QMS?', the conversation frequently falters.

In practice, a vendor's QMS may not accurately reflect the realities of the software development life cycle, especially in the following scenarios:

1. The QMS describes the 'what', not the 'how'. The QMS outlines high-level requirements (the what), but lacks detailed guidance on technical implementation (the how) – as per ICH E6 R3[1].

2. Vendor internal audits focus on validation artifacts, not engineering practices. Vendor audits typically evaluate the completeness of validation documents (e.g., test scripts, reports), while overlooking whether software engineering processes are effective and followed. As a result, there's little to no independent oversight of the core technical practices that should embed and verify quality within the software development process.

3. Customer reliance on vendor testing may be misplaced. Regulated customers often depend on the vendor's testing to assure software quality. Yet vendors may focus on ensuring customer User Acceptance Testing (UAT) compliance and acceptance over ensuring the intrinsic robustness of the software product. As highlighted by the FDA[2], testing alone is insufficient to ensure product quality.

4. CAPA systems may not cover technical process. While vendors may show evidence of CAPA management related to QMS compliance and documentation, given point 1 and 2 above, there is no CAPA activity that address deficiencies in the technical software life cycle processes themselves. Consequently, the QMS does not effectively govern the technical aspects of software production that directly impact product quality.

These observations are based on 18 vendor audits conducted across 2023-2024.

Quasar #170[3] and #171[4] explored how a robust test strategy incorporates a variety of techniques tailored to detect different types of software defects. These techniques span multiple phases of the Software Life Cycle (SLC) and are essential in reducing the risk of issues that could disrupt regulated business operations. The verification/test strategy documents not only show the techniques used but also their application across the software production line.

VERIFICATION/TEST STRATEGY

(Note testing and verification are used interchangeably in this article).

Why is a verification strategy so important? Because software is inherently complex. The FDA's General Principles of Software Validation[2] highlight several factors contributing to this complexity:

- Branching logic allows software to execute different sets of instructions based on different input data, making it difficult to fully understand – even in short programs
- This branching can conceal latent defects, which may only surface under specific conditions during production use
- Testing alone cannot confirm software correctness. A combination of verification techniques is required to enable both prevention and early detection of defects
- Due to its inherent complexity, the software development process must be controlled to avoid issues that may go undetected until late stages or after release.

A QMS may appear compliant on paper, for example, with Annex 11 §4.5 ('appropriate quality management system')[5], ICH E6 R3 ('appropriate system to manage quality')[1] or OECD 17 ('software development governed by a QMS')[6], however it may still be ineffective in practice. A compliant looking QMS does not necessarily ensure high-quality software. If the underlying strategy lacks depth in verification practices, the result may be low software product quality, posing risks such as regulatory deadlines, data integrity issues and security vulnerabilities. Poor software quality is not an unavoidable characteristic of software itself, but rather a reflection of the governing QMS.

Figure 1 illustrates the costs linked to defect remediation across the software production line. The ability to quantify these costs signals an organisation's intent to both reduce them and improve overall software quality.

Figures 2 and 3 extend the costs to the regulatory customer's perspective, highlighting the broader cost impact of software defects. Vendors often overlook how poor software quality imposes significant downstream costs on their customers – costs that go beyond immediate technical fixes and can affect compliance, operations and business outcomes.

Figures 2 and 3 underscore the importance of focusing on software product quality.

Verification performed before coding serves as a defect prevention measure, while verification after coding functions as defect detection. Vendors whose verification strategy combines both preventative and detective approaches are likely to produce higher-quality software with fewer defects and a lower risk of production issues.

The final determination that a software product is validated should be based on evidence gathered through planned, structured activities conducted throughout the software development life cycle (FDA[2]). One of the primary objectives of a vendor audit is to assess whether such evidence exists and is traceable.

> Accurate documentation (information) needs to be meaningful to support these activities, as the personnel who make maintenance changes to the software product may not be involved in the original development. (FDA[2])

The following sections explore how vendors could manage meaningful information that provides a clear indication of their software product quality level.

'Verification performed before coding serves as a defect prevention measure, while verification after coding functions as defect detection.'

FIGURE 1. AGGREGATED COST OF DEFECT REMEDIATION (VENDOR)

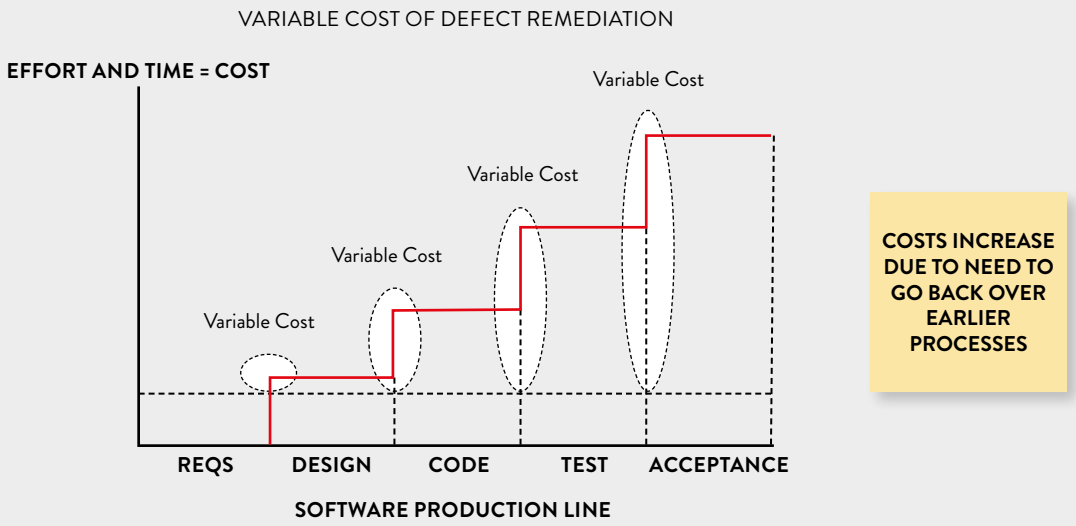


FIGURE 2. MULTIPLIER COST OF DEFECT FOR REGULATED CUSTOMER[7,8,9,10]

PHASE DETECTED	VENDOR COST IMPACT	CUSTOMER COST IMPACT	COST MULTIPLIER (IBM MODEL)	NOTES
Requirements	Low (1x)	Very Low	1 (IBM, Boehm)	Easy to correct during planning. Minimal impact
Design	Low – Medium (3-5x)	Low	3 (IBM, Boehm)	Some architectural rework required
Development/Unit Test	Medium (5-10x)	Low	5 (Boehm, Jones)	Code rework and retesting
Integration Test	High (10-20x)	Medium	10 (Boehm, Jones)	Defects are harder to isolate and fix. Ripple effects
Release/Deployment	Very High (20-50x)	High – Very High	20 (Jones, NIST)	Hotfixes, roll out delays and increased costs
Post release (Production)	Extremely High (50-100x)	Extremely High	50 (IBM, Jones, NIST)	Support costs, brand damage, regulatory exposure (penalties, safety)

FIGURE 3. EXAMPLE OF MONETARY COST OF DEFECT REMEDIATION FOR A REGULATORY CUSTOMER[11]

REAL WORLD EXAMPLE: TOTAL COST OF A COMPUTERISED SYSTEM DEFECT		
DEFECT COST	**VENDOR**	**CUSTOMER**
Find and report defect (1/2 hr)	$0	$40
Vendor costs to remediate	$1,240	$0
Customer costs to correct data arising from defect (8hrs)	$0	$640
Workaround while waiting on fix (1 week's worth of manual workaround)	$0	$16,000
Revalidation and regression tests (10 days)	$0	$6,400
Total	$1,240	$23,080
Loaded Salary Cost $80/hr		

The above figures underscore the importance of focusing on software product quality

SOFTWARE PRODUCTION LINE PROCEDURES AND PLANNING

The software production line comprises a series of SLC processes which, when verified, contribute directly to the overall validation effort. It is essential to document how each production line activity will be performed. This 'how' provides the operational detail needed to ensure a consistent baseline of software quality, regardless of who performs the task, supports effective knowledge transfer between personnel and promotes ease of maintainability, a core attribute of software quality.

Defined acceptance criteria for each production line process output enables measurement of process effectiveness, which is essential for both continuous improvement of the software production line and enhancement of the resulting product quality.

There are more than 80 recognised SLC models, each tailored to address specific development or organisational challenges (Jones[9]).

The foundational software engineering activities of requirements elicitation and analysis, design, implementation, testing and release remain consistent, regardless of the SLC 'flavour'.

As a result, it may be useful to compare the SLC to a manufacturing production line, where the output of one phase becomes the input to the next. Just as in manufacturing, each phase output in the software life cycle should be verified to meet minimum quality standards before progressing to the next phase. This practice minimises the risk of compounding defects and helps reduce the cumulative cost of defect remediation (as illustrated in Figures 2-4).

> The final conclusion that the software is validated should be based on evidence collected from planned efforts conducted throughout the software life cycle. (FDA[2]).

It is therefore considered good practice to assess both the breadth and depth of verification activities applied at each stage of the software production line, as well as the rigour with which they are executed. The more varied and well-integrated the verification techniques, the greater the chance of preventing defects early or detecting them sooner, thereby improving overall software product quality. (See Quasar #170[3] and #171[4].)

A distinguishing feature of a mid to high-tier vendor QMS is its ability to measure the baseline quality of outputs at each stage of the software production line.

Auditor hint

The production line analogy helps illustrate the sequence of logical software engineering tasks required to implement a requirement within a software product. In pharmaceutical drug manufacturing, quality checks are not left until the end of the process, as doing so would incur the highest remediation costs. The same principle applies to software development. Quality should be built in and verified continuously.

In practice, software production lines are iterative and incremental, meaning these processes repeat and evolve over time (as illustrated in Figure 6).

'Just as in manufacturing, each phase output in the software life cycle should be verified to meet minimum quality standards before progressing to the next phase.'

FIGURE 4. SUMMARY OF SOFTWARE LIFE CYCLE METHODOLOGIES AND THEIR YEAR OF ORIGIN. REFER TO QUASAR #139 ON THE RQA WEBSITE FOR MORE INFORMATION

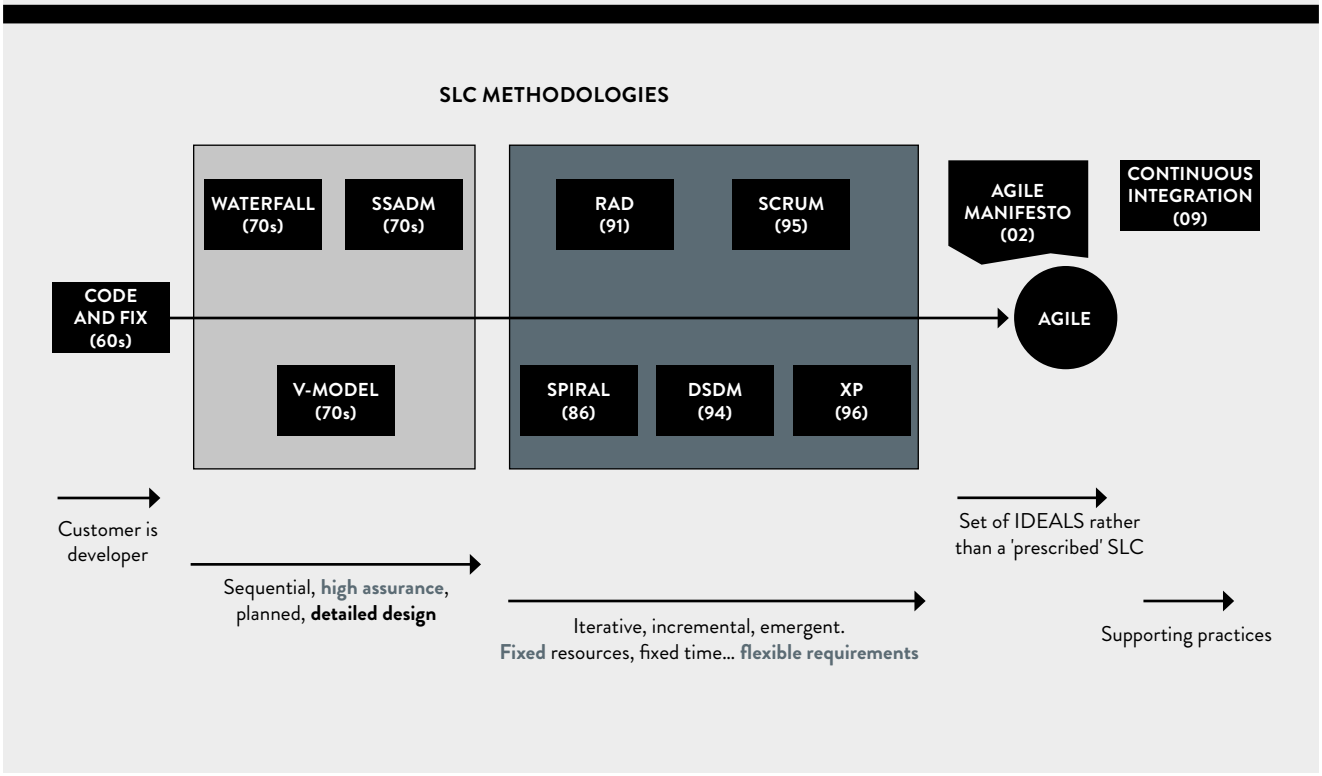


FIGURE 5. LOGICAL SLC AS A PRODUCTION LINE, WITH PROCESS VERIFICATION TASKS ALIGNED TO TEST/VERIFICATION

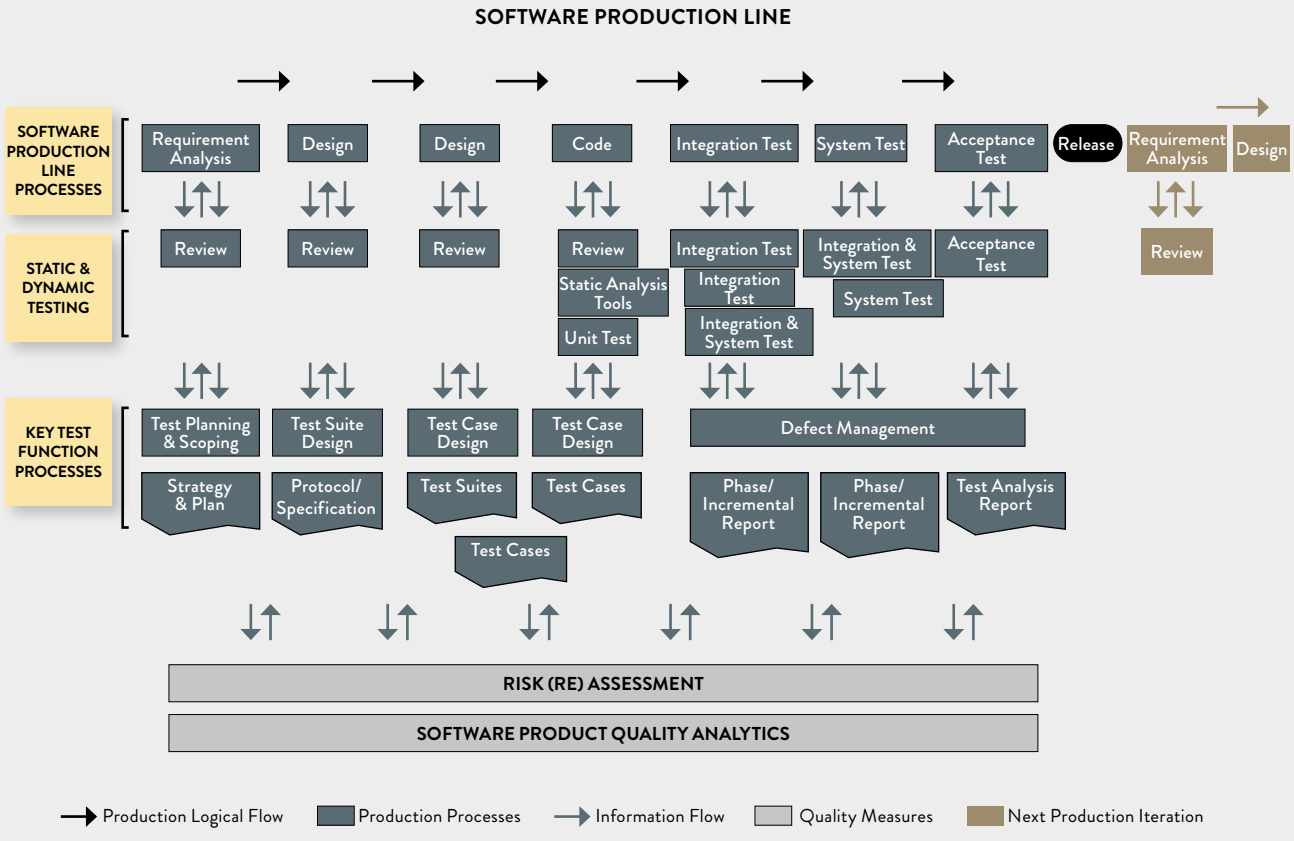
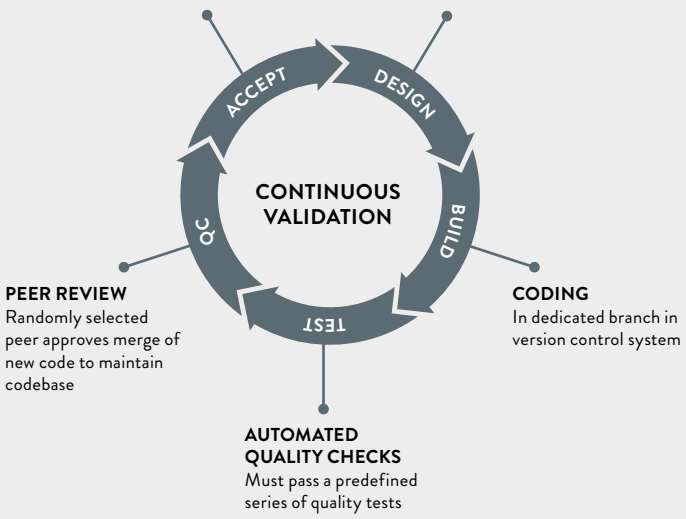


FIGURE 6. A VENDOR'S HIGH LEVEL VIEW OF THEIR ITERATIVE SOFTWARE PRODUCTION LINE



Step	Testing/verification
1. Design	Define the requirements Design review, specification of all likely logic flows Define acceptance criteria
2. Build	Write code changes on separate branch (copy) Write automated tests for acceptance criteria Write automated tests for underlying dependent logic
3. Test	Pre-delivery checks: • Application build from scratch (automated) • Automated tests must pass • Standard quality controls must pass • Coding standards (automated check)
4. QC	Peer review of work: • Review requirement and delivered code • Confirm all testing passed • Assess technical implementation – robust, sustainable and secure If good, approve change (merge to main dev. branch)
5. Acceptance	Product owner reviews delivered change: • Satisfies requirement • Look and feel • Usability IF accepted THEN Done ELSE return to step 1 with feedback

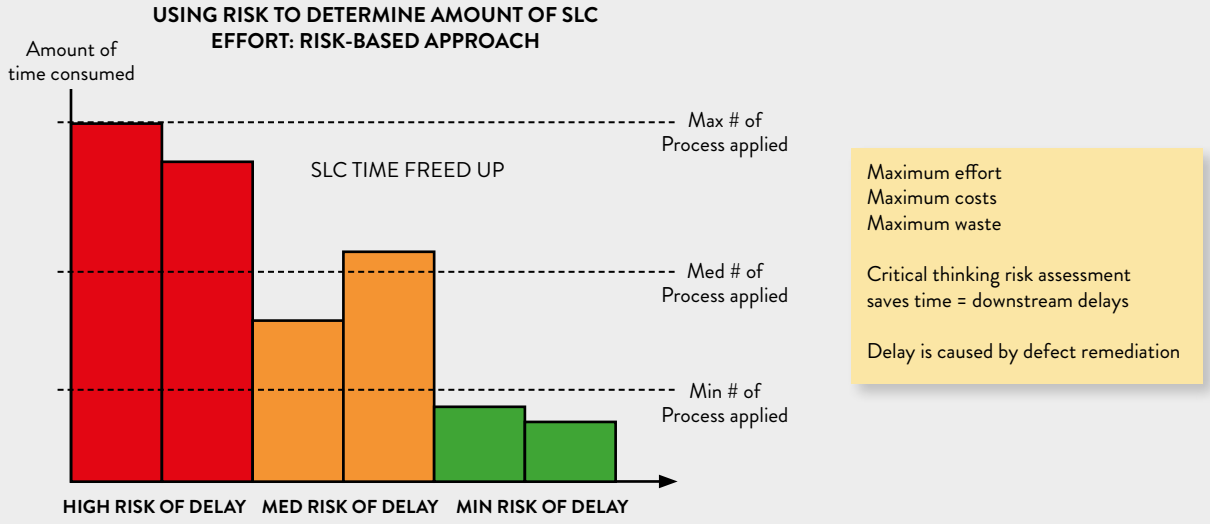
VERIFICATION WITHIN THE SOFTWARE PRODUCTION LINE

Typical verification activities that 'should' be visible before the independent test phase, include.

TABLE 1. SOFTWARE PRODUCTION LINE PHASE

SOFTWARE PRODUCTION LINE PHASE
Requirements: It is not possible to validate software without predetermined and documented software requirements (FDA[3]). This phase focuses on analysing, identifying and defining the information necessary to describe the software product and its intended operational use. The **requirements elicitation** process may include, but is not limited to, the following elements: • Expected **inputs** and **outputs** (data) for each system or feature • **Business functions** the software is intended to perform • **Interfaces** with other systems (inbound and outbound) • **Security requirements**, including access controls and data protection • **Performance criteria**, such as throughput, response time and concurrency • **Error handling** and **fault tolerance** capabilities • Requirements for **user documentation** and technical manuals • **Safety considerations** relevant to system operation • **Regulatory compliance needs**, e.g. 21 CFR Part 11 • **Human–Computer Interaction** (HCI) needs, including usability and accessibility. Capturing and defining these requirements clearly is essential to ensure that the software can be developed, verified and ultimately validated against its intended use.
Risk (Re)Assessment: Identification of technical and operational risk scenarios, including risk quantification, implementation of risk mitigation measures and verification that those mitigations are effective – all forming part of a risk-based verification strategy.

FIGURE 7. RISK-BASED APPROACH APPLIES BREADTH AND DEPTH OF SLC PRACTICES, PARTICULARLY VERIFICATION PER REQUIREMENT RISKS



EXAMPLE VERIFICATION TECHNIQUES (REFER TO QUASAR #171[4] FOR DETAILS ON THE VERIFICATION TECHNIQUES)	AUDIT
[Verification technique] Requirement inspection: Requirements should be reviewed to ensure they align with ALCOA+ principles (Attributable, Legible, Contemporaneous, Original, Accurate, plus Complete, Consistent, Enduring and Available). The inspection should verify that each requirement is clearly identified, accurate, complete, consistent (both internally and across requirements), unambiguous, measurable and testable. [Verification technique] Initial test case design: Test case development should begin in parallel with requirements analysis. This early activity helps identify conflicts, gaps, ambiguities, complexity and potential risks or errors in the requirements. By doing so, it reduces the likelihood of false assumptions being embedded into the design and build phases. As a result, it minimises downstream defects, shortens the testing cycle and reduces the effort required for defect remediation, retesting and regression testing.	[Audit activity] Review a sample of requirements for clarity and testability. [Advanced audit activity] Discuss the requirement elicitation and analysis process. Ascertain if there are any metrics or quality indicators used to measure the effectiveness of the requirement process and its outputs. [Advanced audit activity] Ask how requirements are confirmed to be fully specified and appropriate prior to design. Explore how design is updated when the requirement changes. Ask for visibility of this approach. [Advanced audit activity] Walkthrough of the software production line from a definition though to implementation, testing and release.
[Verification technique] Formal review by several roles.	[Audit activity] Verify that technical risk mitigations are actively incorporated. For example, confirm the use of mirrored (RAID) disk storage to mitigate the risk of hard drive failure. The effectiveness of this mitigation should be tested, with risk reassessed to determine if it has been reduced to an acceptable level. [Audit activity] Check whether risk is reassessed at each software production line process output, as each stage generates new information that could affect risk status. This helps determine if risk management is a practical, integral tool throughout the process or merely treated as documentation without real impact. [Advanced audit check] Evaluate the approach to risk mitigation and verification beyond simply confirming that requirements function as intended. Discuss when and how verification planning was conducted at this point, including the range and scope of technical techniques employed to minimise the risk of defects reaching production.

SOFTWARE PRODUCTION LINE PHASE

Design: Requirements define the set of problems that the software design must address (Swebok[12]). Design is an iterative, multi-layered process that outlines how the software product will be constructed. Its objectives include:

- Preventing data errors through careful data flow control and well-defined data structures (FDA[13])
- Reducing complexity, which is critical for both general software quality (FDA[2]) and especially for safety- and mission-critical systems (McConnell[14])
- Providing clear guidance to developers and implementers on how to build the system (Swebok[12])
- Establishing the basis for a comprehensive test approach and the development of test cases to verify that the design is correctly implemented and functions as intended (Swebok[12]).

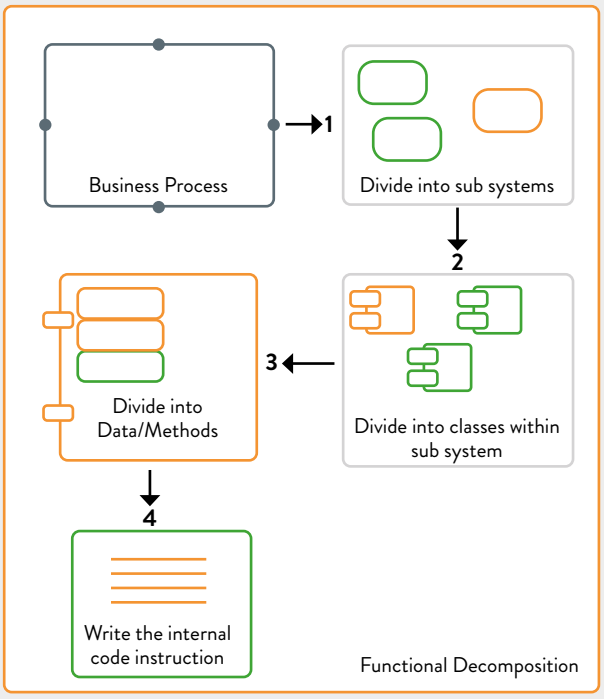
Effective design also emphasises ease of testability and maintainability, facilitating future modifications and supporting ongoing verification and validation efforts (Quasar #159[15]).

High Level Design (for example):

- Hardware
- Logical structure
- Functions and interfaces between components
- Data control flow design
- Data 'structures' design (input, output and internal data structures)
- Data storage dimensioning
- Event logging, error handling and recovery
- Security architecture
- Help system architecture.

System design principles may include modularisation, abstraction, encapsulation, separation of the GUI and logic, high cohesion, loose coupling, uniformity, verifiability.

FIGURE 8. DIVIDE AND CONQUER DESIGN APPROACH



Database intended design versus implementation.

FIGURE 9. DESIGN OF DB (DATA FLOW STORE)

```
1 //3rd Normal form PC Table
2 Column   |Data Type |Notes
3 TAGNUM   |Char(5)   |Primary key
4 COMPID   |Char(4)   |Foreign key - > COMPUTER. COMPID
5 EMPNUM   |Decimal(3)|Foreign key - > EMPLOYEE. EMPNUM
6 LOCATION |Char(12)  |Check constraint: must be 'lab'
```

Figure 9 illustrates the design of a database store (Data Flow Store), where the COMMPID field links to the COMPUTER table containing computer details and EMPNUM links to the EMPLOYEE table that stores the employee information.

EXAMPLE VERIFICATION TECHNIQUES (REFER TO QUASAR #171[4] FOR DETAILS ON THE VERIFICATION TECHNIQUES)	AUDIT
The vast majority of software problems are traceable to errors made during the design and development process. (FDA[2]).	Careful attention to software architecture – such as employing a modular design – during the design stage can significantly reduce the scope and effort of future validation activities when software modifications become necessary.
[Verification technique] Review/walkthrough (FDA[2]) of design, such as: • Process control flow • Data flow • Complexity • Security • Maintainability • Memory allocation • Sizing and capacity planning. Test case design based on design content such as scalability tests, performance tests. Quality mature vendors conduct themed inspections targeting specific areas of interest, such as performance, fault tolerance or security.	[Audit activity] Verify traceability of design elements back to the original requirements. Confirm that risk reassessments have been performed as needed. [Audit activity] Check for evidence of design standards and adherence to same. [Audit activity] Ensure that data flow diagrams and data definitions are clearly specified. [Audit activity] If a relational database is used, ensure to confirm the database design was established prior to implementation. This is essential to ensure data integrity constraints are properly incorporated, helping to prevent: • Data inconsistencies • Application performance issues. [Advanced audit check] Many database systems can generate a representation of the implemented database schema, but this is not a substitute for the pre-implementation design process that requires critical analysis. Ask the auditee to provide documentation of data definitions created before the design phase. Inquire how referential integrity was planned and designed prior to database construction. [Advanced audit activity] Ask for evidence of design as an input to test case design. For example for error and exception handling scenarios.

SOFTWARE PRODUCTION LINE PHASE

FIGURE 10. DATABASE MANAGEMENT SYSTEM SCHEMA

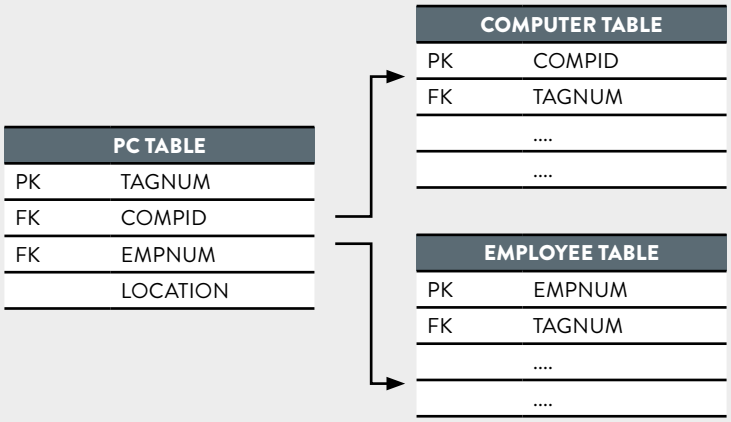


Figure 10 shows the implemented DB Schema (storage). The schematic represents the normalised database structure, maintaining referential integrity: where the foreign key in one table corresponds to the primary key of another table (Pratt & Adamski[16]).

Low-level Design (for example)

- Key software component algorithms
- Software component data structure
- Interface communication protocol (data message).

Code and Unit Test

The translation of design into programming language(s) represents the final stage of breaking down requirements into executable code.

Various tools are available to enforce coding standards, ensuring consistency in clarity, commenting, indentation and overall code structure.

FIGURE 11. SQL CODE FOR FIG 10. DESIGN

```
1 CREATE TABLE PC
2       (TAGNUM      CHAR (5),
3       COMPID       CHAR (4),
4       EMPNUM       DECIMAL(3),
5       LOCATION     CHAR(12)
6       CHECK (PC. LOCATION IN ('LAB'))
7       PRIMARY KEY      (TAGNUM)
8       FOREIGN KEY      (COMPID) REFERENCES COMPUTER
9       FOREIGN KEY      (EMPNUM) REFERENCES EMPLOYEE    )
```

EXAMPLE VERIFICATION TECHNIQUES (REFER TO QUASAR #171[4] FOR DETAILS ON THE VERIFICATION TECHNIQUES)	AUDIT
[Verification technique] 1/3 presentations (group walkthrough), inspection or informal review (via email/messenger tool with information retained). Formal code review for less experienced team members.	[Audit activity] Check for traceability back to high level design/ risk assessment.
[Verification technique] (In)formal code review for consistency, style, adherence to (defensive) standards, errors, complexity and ease of maintainability. [Verification technique] Code walkthrough for design flaws, defects and ease of maintainability. Typically (when used) they are applied to critical code (FDA[3], Quasar #171[4]). [Verification technique] Use of static analysis tools to assess, for example, clarity, conventions, complexity and security. [Advanced quality assurance] Use of defensive programming standards. (Quasar #159[15]) [Verification technique] White box unit testing of the code logic over the testing of the feature level.	[Audit activity] Ensure that the higher risk features are traceable to the code and that the associated code has been subject to an appropriate review technique. Audit experience indicates that code reviews are often informal, focusing mainly on code structure and style, and heavily reliant on the reviewer's expertise. **note unit testing is reviewed as part of the code review. [Audit activity] Ensure that compiler warnings are checked – at a minimum – for release candidates. Code reviews should not approve code containing unresolved compiler warnings, as warnings do not block software builds but can lead to defects that may escape unit testing (FDA[3]). [Audit tip] Don't let concerns about technical depth deter you from reviewing code-level activities. Use approachable questions such as, "Can you explain this to me in layman's terms?" or "Show me the traceability for requirement 123 in the design," or "Can you demonstrate the unit test that verifies error handling for invalid data inputs?"

INDEPENDENT TEST

Unit testing marks the transition into white box testing techniques (for further details on testing methodologies, refer to Quasar #171[4]).

Testing is a key focus during audits because it is typically the one verification activity that is performed at some level. Consequently, testing can often provide valuable metrics related to software product quality, particularly defect identification.

Quasar #170[3] defined defects as, amongst others, non-conformances to requirements. Defects are of interest because they can provide an insight into the software product quality, derived from the software production line (QMS) processes.

For those interested in deepening their understanding of software production Life cycle activities, the RQA offers an introductory training course on Computer System Validation (CSV) twice annually. More information is available on the RQA website *www.therqa.com/learn-develop-connect/courses-and-events/events/course/introduction-to-computer-systems-validation*

TEST STRATEGY

There is no single test technique or phase that can ensure that a software product has been thoroughly tested (FDA[2]). The absence of defects during testing should not be interpreted as proof that the software is defect free. That is why audit discussions are looking for more than just 'acceptance' level test objectives.

As per the previous articles in this series, an effective strategy involves an aggregation of various techniques across multiple test phases. This layered approach strengthens software product quality and reduces the risk of operational issues.

A test strategy outlines the vendor's approach to verification activities, specifying the techniques applied at key stages of the software production line (see Figure 5), the required levels of independence and serving as a guiding framework for the organisation. For further details on test strategy components, refer to Quasar #170[3] and Quasar #170[4].

TEST PLAN

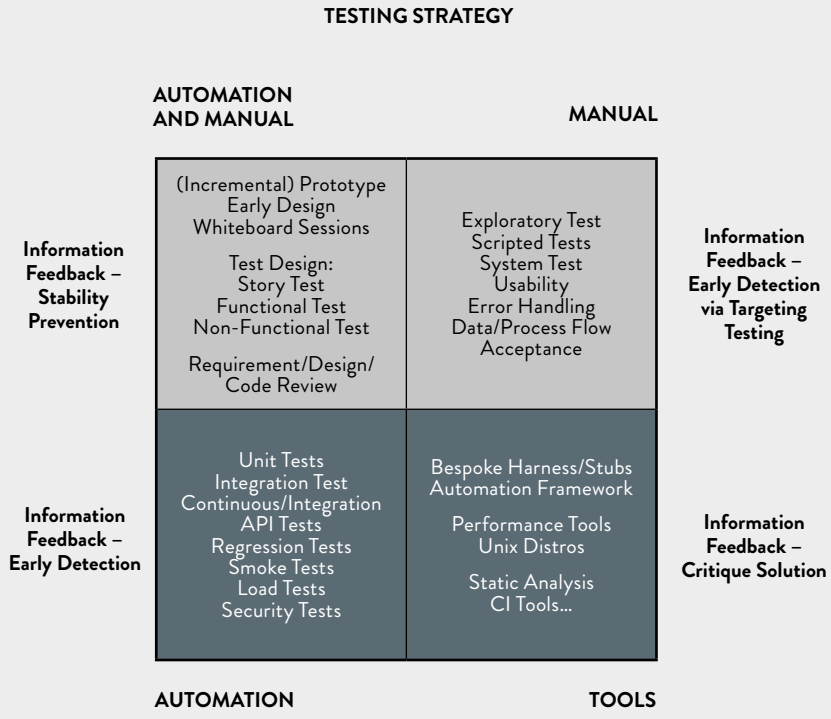
While the test strategy defines an organisation-wide, high-level approach to testing, the test plan provides a detailed, project- or release-specific roadmap for how testing will be carried out. The strategy serves as the blueprint to achieve the highest software product quality, whereas the plan tailors activities to the particular needs of a given release.

Effective test planning should outline tasks, defect and configuration management processes, resource allocation, review activities and potential project risks that could delay testing or lead to defects being released.

Several resources offer guidance on the structure and content of test documentation:

- IEEE 829 Standard for Test Documentation (IEEE17) – now deprecated but still accessible online
- IEEE 29119-3 Test Documentation Standard (IEEE18) – the current replacement for IEEE 829.

FIGURE 12. TESTING STRATEGY EXAMPLE



A quality-focused vendor testing approach aims to deliver effective and efficient testing within the constraints of time and budget, providing meaningful insights into software product quality. Key attributes of such an approach include:

- Prioritising test execution when the code is ready
- Emphasising test process efficiency
- Applying a software engineering mindset to test cases – writing them once for modular reuse and easy maintenance (e.g., updating a single login instruction test case referenced by many others)
- Employing a variety of testing techniques and methods based on requirements, design and risk considerations
- Focusing on gathering, analysing, and acting upon test information and feedback
- Defining a release-specific test strategy that highlights critical features for regression testing and uses risk reassessment to target the highest-risk areas
- Capturing and analysing test execution progress and metrics
- Maintaining independence from development teams to ensure objectivity.

An example of a test planning approach is the Systematic Test and Evaluation Process (STEP) that leverages the concepts described in IEEE29119[18]:

1) Create the plan: objectives, scope, test approach, environment, resources.
2) Acquire the test-ware (test-ware – all of the artefacts and resources required for the test activity):
   a. Inventory of the test objectives (requirements, design, implementation).
   b. Design the tests (architecture, environment, requirements, design and implementation based, inputs, steps and expected results).
3) Implement the plans and designs:
   a. Prepare test data, setup environment, review readiness.
4) Measure the software behaviour:
   a. Execute tests, log outcomes, raise incidents or anomalies.
   b. Evaluate tests, compare results to acceptance criteria.
5) Report:
   a. Summarise findings, deviations and results.
   b. Conclusion: Evaluate the software product quality, establish feature quality, establish number of defects in production release.
6) Evaluate the test process, store assets and close.

Note: The authors do not follow the complete set of documentation requirements outlined by IEEE but strongly support the critical thinking necessary to produce meaningful documentation content. In practice, several IEEE documentation types can be consolidated into a single artifact – for example, a specification that includes test design, data design and related test case suites. The authors prioritise the quality of information over the sheer volume of documentation.

TEST PROTOCOL/ SPECIFICATION

The key to software test quality lies primarily in the test elicitation phase rather than in the execution itself (FDA[2]). Test design requirements are derived from the test strategy, system requirements, regulations and design specifications. The scope of testing can then be defined using the '5W's +' approach (who, what, where, when, why and how), which helps establish the expected outcomes to be verified and challenged during testing. This line of reasoning is often applied during vendor audits when discussing test scope and coverage.

Key test design attributes outlined by the FDA (FDA[2])*** include:

- The expected test outcome is defined in advance
- Effective test cases have a high likelihood of detecting errors
- A successful test case identifies an error
- Testing is conducted independently from development
- Testing only typical scenarios is insufficient
- Test documentation enables reuse and independent verification of test results during reviews and serves as a baseline comparator if defects arise in later phases or in operation.

** The next article will explore test attributes related to automation.

Audit Check

When engaging vendor testing teams, the authors focus on assessing the quality of testing in addition to the quality of documentation. Specifically, they seek to understand the scope and variety of testing techniques that make up the release's test strategy and how frequently each test objective is executed (once, multiple times) to evaluate the stability of each requirement. This insight may help inform a least-burdensome validation approach.

TEST DATA

Many of the most challenging bugs to detect are data-driven, so identifying appropriate test data alongside the right test techniques can significantly enhance defect detection within a limited timeframe. Given the inherent complexity of software (FDA[2]), test data should be carefully defined to drive test scenarios effectively.

Selecting test data depends on a clear understanding of data flow and data definitions, which means that the relevant requirements and design documentation (covering data flow and data definitions) must be in place and approved. The process of eliciting test data adds an additional layer of design review.

Test data plays a critical role throughout the various testing phases and activities outlined in Quasar #170[3] and Quasar #171[4], enabling:

- Modularisation of test cases, where the same test case instructions are reused with different test data to generate varied results
- Keyword-driven automation testing, where test flows are driven by test data, aligning with automation design paradigms
- Comprehensive coverage of data types, including positive and negative scenarios, error handling and defect detection
- Reduction of the risk of production issues caused by omitted test data values or types, as thorough test data coverage inherently spans the range of production data
- Demonstration of a high level of software production line quality maturity.

Defect reports should include precise definitions of the test data used, facilitating efficient retesting to reproduce defect scenarios and enabling faster root cause analysis and remediation.

'The key to software test quality lies primarily in the test elicitation phase rather than in the execution itself (FDA[2]).'

TEST CASES

There are numerous methods and tools a vendor can use to document test cases, which is beyond the scope of this article. What matters most is that the information provided is appropriate for the specific system under test. The level of detail often requires balancing trade-offs. Detailed, step-by-step test cases assume less tester knowledge, making knowledge transfer easier when testers change and improving reproducibility. Conversely, high-level test instructions reflect the experience of the testers and should be considered when engaging with vendors.

Frequently, the authors see that vendor's test cases mimic the regulatory industry's typical style and scope, often focusing on user acceptance testing. When questioned, vendors often explain this approach is driven by auditor expectations. To satisfy auditors, vendors tend to emphasise verifying test documentation content and appearance. However, a deeper review may reveal that the same test scenario is repeated at multiple levels (and in lieu of) unit testing, informal testing (to confirm explicit test steps) and formal testing (to produce the official documentation). In such cases, considerable time and effort are spent on documentation style rather than on executing tests designed to detect defects early, preventing them from reaching production.

As effort is directed toward perfecting test documentation, testing ensures the system performs as expected for specific test objectives. However, less time is available to verify that the system does not perform unintended actions by applying diverse test techniques across different test phases. Consequently, if end users operate outside documented test scenarios, latent defects may surface.

Is this approach truly effective against risks associated with computerised systems, such as data integrity errors or delays in regulatory submissions due to hidden defects? The authors advocate shifting some focus away from documentation quality toward the actual software product quality – since product quality ultimately determines operational risk, business impact and patient safety.

> 'To satisfy auditors, vendors tend to emphasise verifying test documentation content and appearance.'

What is needed is sufficient test case information to ensure:

- Effective knowledge transfer to stakeholders at later stages
- A reliable control test for future defect or regression testing, aiding root cause analysis
- Ease of maintenance. Well maintained test cases reduce human error and administrative burden. Adopting programming best practices, such as 'write once, reuse multiple times', using clear objective statements, employing variables or placeholders for test data and applying self-explanatory test case identifiers can help achieve this
- Design readiness for future test automation
- Provision of data that supports advanced quality activities like software quality analytics and reporting.

FIGURE 13. PROBLEM OF AN ACCEPTANCE LEVEL ONLY TEST FOCUS

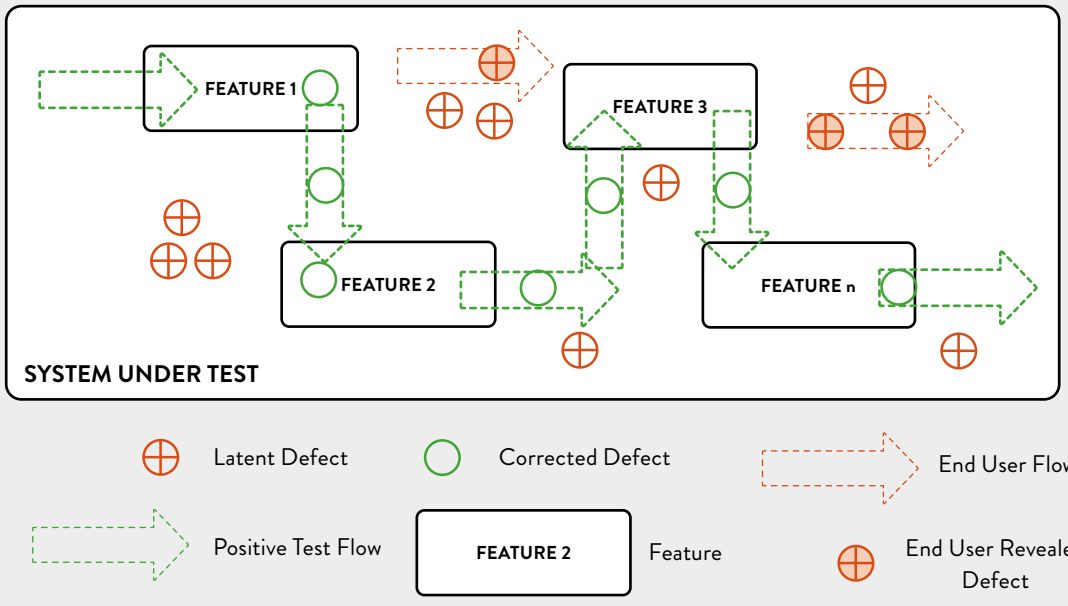


FIGURE 14. WHAT DOES VERIFICATION ACTIVITIES TELL US ABOUT SOFTWARE PRODUCT QUALITY?

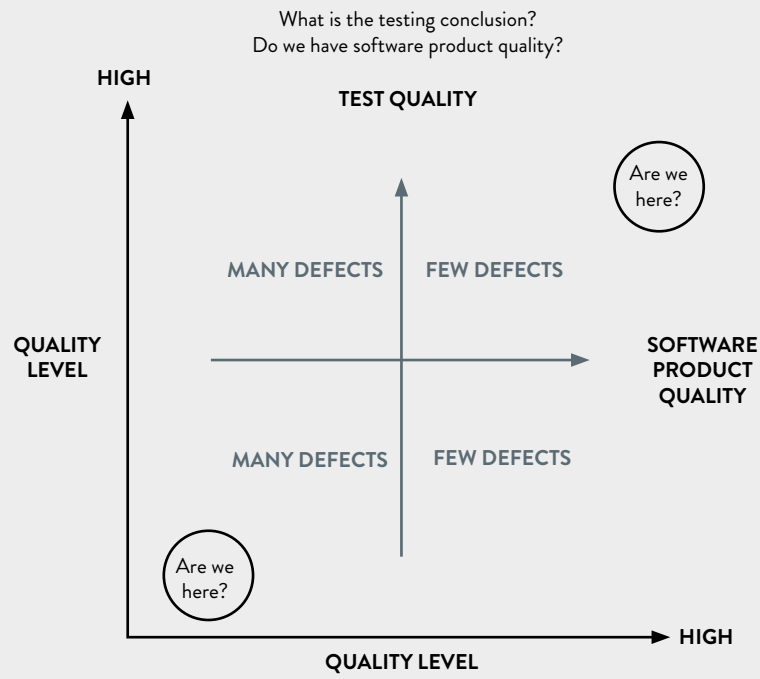


FIGURE 15. STATIC VERIFICATION MEASUREMENTS: DEFECT PREVENTION

ITEM	MA	MI	Q	I	COMMENT FOR PROJECT SUMMARY/REVIEW
Features	13	22	12	16	(Ma)jor and (Mi)nor defects were required to be remedied prior to approvals. (Q)ueries and (I) improvements were agreed between requirement owners and reviewers
User stories	4	6	4	9	Non-functional deficiencies
HLD	5	3	16	16	N/A
Dev plan	1	6	10	7	Missing Review Action List and Information Form Also version signed is 0.4 and not 0.10
Test plan	-	1	-	-	Incorrect schedule duration
Test spec	0	9	1	14	Needs another review after initial review updates
Code review	-	-	-	-	Measurements of deficiencies were not captured. Not following QMS review template. Needs to be enforced in projects

SOFTWARE QUALITY ANALYTICS

A vendor test report stating that all planned tests were completed and all defects were reviewed and deemed acceptable for release does not, by itself, provide meaningful insight into the actual quality of the software product. Context is essential to understand what constituted the 'planned' testing. Simply listing test case identifiers often fails to convey the scope and depth of the testing challenges, as highlighted in Quasar 170[3] and Quasar 171[4].

So what could meaningful information look like? The following examples are extracts from software quality analytics conducted per vendor audits to highlight what information could be provided/generated.

INTERNAL VENDOR 'A' (PREVENTATIVE FOCUS)

Vendor 'A': Clinical Trial Management System. Static verification activities were piloted to assess the effectiveness of the production line processes in driving improvement initiatives and to evaluate whether these static checks reduced defect volumes in later testing phases. The static verification efforts yielded the following results: Across the requirements and HLD (High Level Design), a total of 53 defects were prevented from being built into the software product. This enhanced quality by minimising false assumptions coded into the system, reducing ripple effects caused by software branching and decreasing delays associated with fixing and retesting defects in subsequent test cycles. The analytics clearly showed that quality was proactively built into the system.

'… software validation is a matter of developing a 'level of confidence' that the device meets all requirements and user expectations' (FDA[2])

'The static verification efforts yielded the following results: Across the requirements and HLD (High Level Design), a total of 53 defects were prevented from being built into the software product'.

VENDOR 'B': PREDICTING THE SOFTWARE PRODUCTION QUALITY

The software production line employed a risk-based approach, prioritising focus on processes with higher operational and technical risks. Both static and dynamic verification activities were integrated throughout the iterative and incremental production line. Hand-offs (HOs) transitioned work into the independent testing phase, where a risk-based strategy was applied – testing the highest-risk features first and revisiting them continuously across subsequent HOs. This 'test early, test often' approach emphasised establishing quality levels for the system's most critical features.

The volume of defects raised fell drastically after the third release into the test function. The figures were used to predict the software quality of the release.

At the end of HO2, HO1 detected 62 out of 77 known defects (80%). By HO6, HO5 had found 96% of the total defects (86). Extrapolating, 100% would equal approximately 89 defects, leading the project team to estimate that three defects remained at the end of HO6. Actual system use in operations over three months revealed four minor defects.

Additionally, defect counts from HO4 to HO6 were consistent and of low criticality, supporting the decision to cease testing.

'Software verification and validation are difficult because a developer cannot test forever, and it is hard to know how much evidence is enough.' (FDA[2])

VENDOR 'C' CTMS AUTOMATION TOOL: SIMPLE DEFECT TRENDING

The vendor's QMS demonstrated a strong level of detail regarding software production line processes. However, the accompanying test report lacked meaningful analytics. It simply stated that 'all planned tests were executed and results were accepted', without further context or insight.

The audit team accessed the defect management tool data during the audit. Some of the analysis performed revealed a downward trend in overall defect volume over a 30-month period, suggesting continuous improvement in both software product quality and the underlying QMS processes.

Another example was the analysis of defect trends across key product features (anonymised to reduce risk of product/vendor identification).

FIGURE 16. DEFECTS FOUND PER SYSTEM TEST ITERATIONS

	HO1	HO2	HO3	HO4	HO5	HO6	TOTAL DEFECTS
Defect Totals	62	15	1	2	3	3	86
DDP %	-	80.5	98.72	97.5	96.39	96.51	-

FIGURE 17. DEFECT TOTALS ACROSS 30 MONTHS

SYSTEM TEST DEFECTS PER YEAR		
2022	2023	2024 (6 months)
162	110	43

FIGURE 18. SIMPLE DEFECT TRENDING TO INDICATE FEATURE QUALITY ACROSS 30 MONTHS

TREND: DEFECTS PER YEAR			TREND: DEFECTS PER YEAR		
FEATURE	TOTAL	PERCENT	2022	2023	2024
A	136	43%	70	39	27
B	13	4%	4	8	1
C	15	4%	9	5	1
D	11	4%	7	3	1
E	20	6%	10	10	0
F	59	19%	36	18	5

While most features showed a decline in defects, Feature B experienced a 100% increase in defects in 2023 and Feature E remained unchanged compared to 2022.

These insights prompted audit discussions in which the vendor committed to investigating and addressing the spike in Feature B and reassessing the stagnant trend in Feature E.

As part of the outcome, this data-driven analysis was shared with the vendor and the regulatory customer to discuss a risk-based validation strategy. Improved quality features such as Feature D would be considered for a reduced validation approach in future releases. In contrast, Feature B would undergo full validation and Feature A would be earmarked for reduced validation pending further positive trend data.

The vendor committed to enhance future release reporting by including software product quality analytics, including the breakdown of feature risk, aggregated test coverage and defect criticality analysis, to aid the validation effort for the regulated customer. This response aimed to support a least burdensome validation effort for the regulated customer while maintaining the software quality oversight.

'The capability to monitor and detect performance issues or deviations and system errors may reduce the risk associated with a failure of the software to perform as intended and may be considered when deciding on assurance activities.' (FDA[2])

AUDIT ANALYTICS

A key aspect during vendor audits is to assess the quality level of the software product being released and to assess whether this quality has improved over successive releases. In many cases, vendors either do not perform this analysis or lack mature measurement practices to provide meaningful insights. In such cases, when conducting follow-up audits, the authors will draw on the regulated customer's internal metrics. Software quality analytics can uncover valuable insights into the software production processes (i.e. the QMS) that contribute to the overall product quality.

'A conclusion that software is validated is highly dependent upon comprehensive software testing, inspections, analyses, and other verification tasks performed at each stage of the software development life cycle' (FDA[2]).

Vendor discussions typically focus on the need to provide quantifiable, meaningful information. Quantitative data offers insight not only into the software product's quality for a given release but also into the performance of the underlying production processes and the maturity of the vendor's QMS. Vendors with more advanced quality systems are generally able to produce this level of information across each verification stage of the software production line.

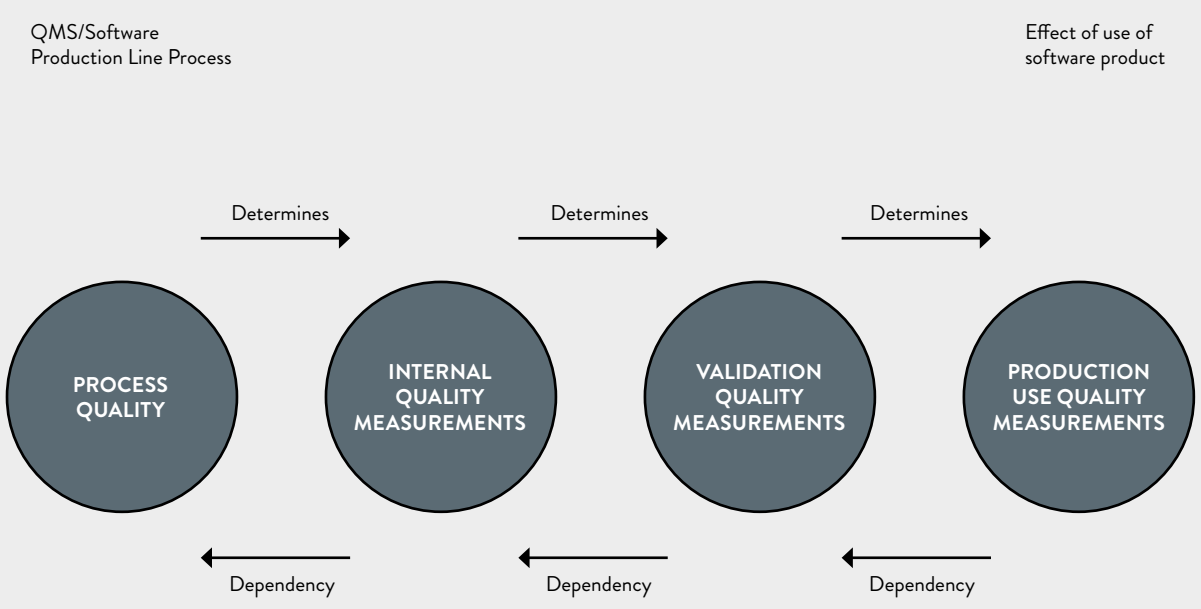
'Software quality analytics can uncover valuable insights into the software production processes (i.e. the QMS) that contribute to the overall product quality.'

CONCLUSION

The QMS governs the software production line – the structured set of processes designed to deliver a high-quality software product. The ultimate goal is to produce software that performs reliably for the end user. However, the belief that simply documenting a process guarantees quality is a misconception: 'The assumption is that a documented process equates to quality, but this is a fallacy.' (McDowall[20]).

'Measures such as defects found in specifications documents, estimates of defects remaining, testing coverage, and other techniques are all used to develop an acceptable level of confidence before shipping the product.' (FDA[2])

FIGURE 19. ISO91261: RELATIONSHIP BETWEEN THE SLC QUALITY PROCESS AND SOFTWARE PRODUCT QUALITY MEASUREMENT



Software product quality is a key differentiator between poor, good and excellent vendors. See Figure 20.

FIGURE 20. THE MORE COMPREHENSIVE AND MEASURABLE THE SOFTWARE PRODUCTION LINE (QMS) PROCESSES, THE HIGHER THE RESULTING SOFTWARE QUALITY - OFTEN REFLECTED IN DEFECT REMOVAL EFFICIENCY (DRE)

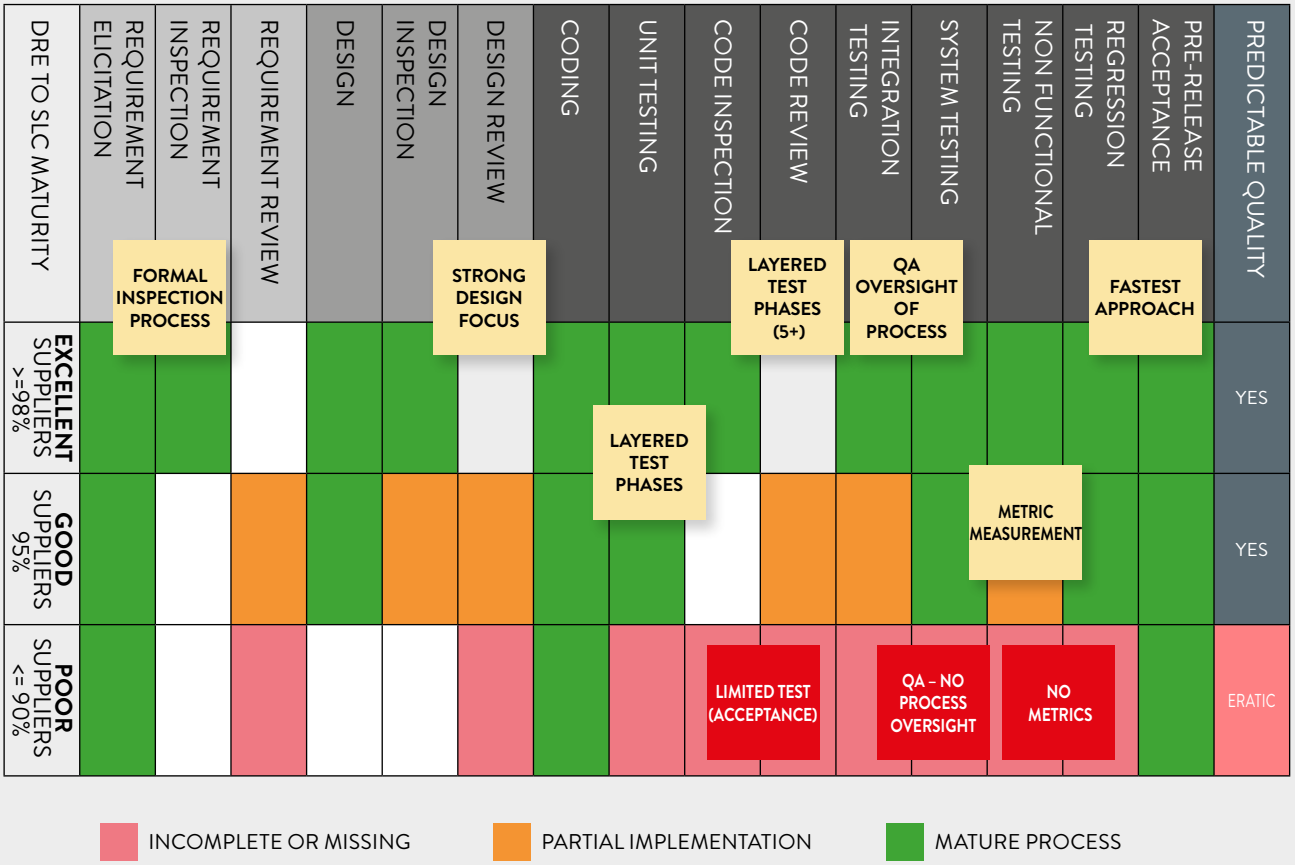


FIGURE 21. SOFTWARE PRODUCT QUALITY MEASUREMENT: ISO, CMMI AND FDA

ISO 9001/90003	CAPABILITY MATURITY MODEL INTEGRATION (CMMI v2.0) – (2018)	FDA THOUGHTS
• ISO 9001 1994: appropriate sources of information...to... analysis and eliminate nonconformities (failure) • 4.20 Statistical Techniques (for quality management) • ISO 9001 2015 9.1.3 Monitoring. Measurement, Analysis and Evaluation • Conformity of (software) product • Measure the performances and effectiveness of the QMS (software production line) • (Application of 9001 to software) ISO 90003 2018: 9.1 Monitoring, Measurement, Analysis & Evaluation • Analysis of root cause of non-conformities...(as) input to Preventative Action (PA)... reverse unfavorable trends in metric levels may be considered as PA.	Apply management via quantitative analysis 5 levels of quality maturity: 1. Initial: none 2. Managed: measurement and analysis 3. Defined: process instruction 4. Quantitatively managed: process performance 5. Optimising: caused analysis and resolution	General Principles of Software Validation 2002 3.1.2 Measures such as defects found in specifications documents, estimates of defects remaining, testing coverage and other techniques are all used to develop an acceptable level of confidence before shipping the product.

Quality outcomes are the cumulative result of robust software production processes. Low product quality signals weak or missing practices, such as poor design reviews, lack of code inspections, ineffective static verification and narrow test scope. Conversely, higher product quality correlates with a mature QMS: one that applies preventive and detective verification activities at each stage of the life cycle.

Merely repeating acceptance-level tests to produce compliant documentation adds little value to the industry. Somewhere along the way, the core intent may have been diluted.

The principle of actively measuring product quality during production dates back to foundational quality thinkers like Shewhart (1939) and Juran (1951) (UKEssays[21]), and remains central in current frameworks such as ISO, CMMI and FDA guidance. These measurement practices are applicable to all software production lines, regardless of development model.

The active measurement of software product quality throughout the production process is as old as the Quality Management System approach itself: from Shewart (1939) and Juran (1951) (UKEssays[21]) to 'current' ISO, CMMi and FDA thinking.

As measurement is applicable to any software production line, it provides a mechanism to compare one vendor to another.

'When you can measure what you are speaking about, and express it in numbers, you know something about it; but when you cannot measure it, when you cannot express it in numbers, your knowledge is unsatisfactory and you have scarcely advanced in your thoughts beyond the state of science.' (Kelvin[22])

REFERENCES

1. Guideline for Good Clinical Practice E6 (R3), ICH, 2025
2. General Principals of Software Validation, FDA, 2002
3. Software Testing: Measuring Vendor Software Quality – Part One, O'Neill, McManus, Quasar #170, RQA, 2024
4. Software Testing: Measuring Vendor Software Quality – Part Two, O'Neill, McManus, Quasar #171, RQA, 2025
5. Annex 11
6. OECD 17
7. IBM Technical Report (referred to in various IBM papers and books, though original report is not publicly archived)
8. Software Engineering Economics, Boehm, B.W., Prentice-Hall, 1981
9. Jones, C. Applied Software Measurement: Global Analysis of Productivity and Quality, McGraw-Hill, 2008
10. The Economic Impacts of Inadequate Infrastructure for Software Testing, NIST 2002.
11. Empowerment Quality Engineering Ltd internal data.
12. Software Engineering Book of Knowledge (SWEBOK) v4.0, IEEE, 2024
13. Guidance for industry, Computerised systems used in Clinical investigation, FDA, 2007
14. Code complete 2, A practical handbooks for software construction, 2004, McConnell, Microsoft press
15. Using Defensive Approaches to Build Security into Computerised Systems: Auditor Tips, McManus, Quasar #159, RQA, 2022
16. Database Systems Management and Deign, Pratt & Adamski, 1994, Course Technologies
17. Standard for Test Documentation IEEE-829, IEEE, 2008
18. Standard for Test Documentation IEEE 29119-3, IEEE, 2021
19. Manging in the Testing Process, 2nd Ed, Black, 2002
20. Computer Software Assurance: Perfect Solution or Confidence Trick?, Bob McDowall, Technology Networks, 2024
21. A History of Total Quality Management, UKEssays, 2015, web reference: https://www.ukessays.com/essays/management/a-history-of-total-quality-management-management-essay.php?vref=1
22. Electrical Units of Measurement, Kelvin, PLA, Vol 1, 1883

PROFILES

Barry is a Principal Consultant for Empowerment Quality Engineering Ltd, a Computerised System Regulatory consultancy that bridges the gap between IT and quality.

He focuses on building quality and security into Computerised Systems (CS) by using quality techniques from the wider software industry while ensuring regulatory compliance. He leads GxP CSV compliance and IT Supplier/Service Provider audits across the globe; performs IT supplier's software life cycle process improvement, risk assessments to drive validation strategies, validation projects and tailored training.

Barry has over 27 years' experience in Quality Assurance, Software Engineering and IT Administration with vast technical knowledge of every role and every activity within the CS life cycle; including multiple technologies, development methodologies (traditional and agile), databases and programming languages.

He is a member of the RQA IT Committee, the MARSQA and was a member of the ISPE Data Integrity Project team.

Hugh is VP Operations and Quality at PHARMASEAL International Ltd and an independent computer systems validation consultant.

He is an IT professional with over 35 years of experience of using technology in the pharmaceutical industry, initially as a developer, later an implementer and more recently specialising in compliance.