Barry McManus

Hugh O'Neill

SOFTWARE TESTING: MEASURING VENDOR SOFTWARE QUALITY – PART TWO

TEST STRATEGIES AND TECHNIQUES

This topic grew out of a discussion about software testing techniques versus managing the scope of test documentation. Given that the bulk of testing techniques typically reside within the vendor's domain, the authors aim to explore some aspects of this subject area to prepare the QA function when assessing a vendor's test practices and associated software product quality. This is the second article in the series and looks at test strategies and techniques.


user32.dll
kernel32.dll
class=code
OK
window_title
NULL
class=data
Hello, world!
db "Hello", 0
lcs_naive(const string
result {};
asz = a.size(), bsz
i < asz; ++i)
j < bsz; ++j)
max_match
<= result.len)
< match)
lcs_solution
result.len)

Part 1 of this series discussed how a vendor's documentation quality may not correlate with the software product quality[1]. Our IT auditing travels have regularly shown vendor's:

- QMS missing adequate instruction to perform Software Life Cycle (SLC) process tasks, to attain a baseline level of quality
- Quality function focused on documentation quality over software product quality
- Test function is narrowed to proving that requirements work.

Part 1 discussed how software product quality is the absence of a defect which would either cause the System Under Test (SUT) to stop working or cause it to produce incorrect results[1]. Software defects impact the following software product quality attributes: reliability, maintainability, usability, security, data integrity, fitness for use, conformance to requirements and customer satisfaction.

One of the objectives of testing is to uncover a yet undiscovered error[1]. There is a correlation between software defect identification and software test strategies. Typically, the better the testing strategy, the better the software defect discovery, the better the software product quality.

One of the IT Auditor objectives is to assess the robustness of the vendor's testing strategy to reduce the risk of defects occurring in

WHY IS THIS IMPORTANT?

A software defect is a non-conformance to a requirement. Software is complex due to 'branching', i.e. the ability to execute alternative series of commands, based on different inputs[3].

The testing strategy aim is to seek defects for remediation prior to delivery to the regulatory environment. 'Software verification provides objective evidence that the design outputs of a particular phase of the software development life cycle meet all of the specified requirements for that phase'[2].

- A vendor that verifies the requirements and design will result in less defects being designed into the software solution
- Different verification activities are applied to detect specific defects that otherwise may not be identified. Conversely, an omission of a verification activity in a particular phase of the SLC may miss specific defects that may reach operational use
- The more testing techniques considered, the more likely that more defects will be detected and the more likely the software product quality will be higher.

TEST STRATEGY

The software testing strategy is an area where the IT auditor will investigate to ascertain the testing phases[2] being deployed and the different types of testing techniques within the test phases.

This activity can inform the IT auditor of the (quality) maturity of the vendors SLC processes and of the associated software product quality.

Part 1 looked at the common test phases and their objectives[1]. One of the first items to ask the vendor is about the objective of the test strategy. Typically the response will include the conduct of code review, unit testing, independent testing and acceptance testing. See Figure 1.

Quite often the vendor's unit testing does not reflect code logic testing, but rather the focus is on requirement feature testing to confirm that the requirement is working. However, the lack of a range of testing activities may increase the risk of defects remaining in the SUT that will pass into operational use.

It is important to discover the vendor's definition of their testing activities.

Table 1 provides a summary of some of the common test objectives, along with a brief description. Each objective targets a specific outcome. The diagram is representative of functional, compliance and installation test focus that was discussed during a recent for-cause audit.

FIGURE 1. SYSTEM UNDER TEST: POSITIVE TEST FLOW

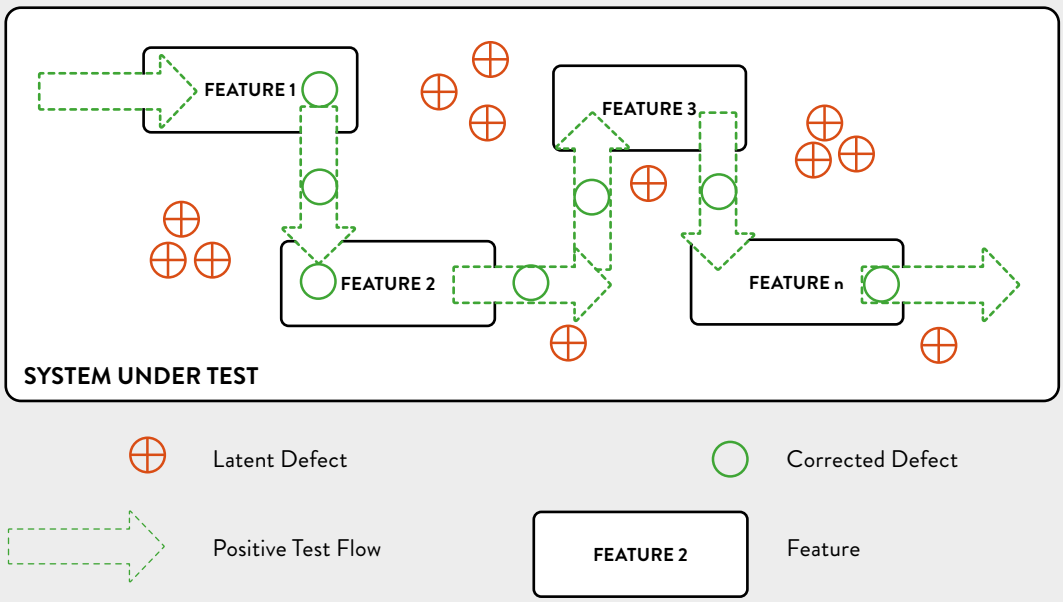


TABLE 1. COMMON TEST OBJECTIVES

TEST OBJECTIVE	DESCRIPTION
Functional	• Determines if the functional requirements of the SUT have been met, including defined standards, specifications, requirements and best practices • In other words, 'the system will do what it is supposed to do'.
Compliance	• Verifies whether the SUT meets legal or regulatory requirements, e.g. 21 CFR Part 11.
Installation	• Verifies the SUT in its target environment, ensuring proper setup, hardware compatibility and operational constraints.
Error Handling	• Assesses how the functionality manages error conditions from system crashes, to logging and reporting, through to preventing further harm • Often called negative or destructive testing • 'Will the system not do what it is not supposed to do'.
Regression	• Re-executes previous tests to ensure that modifications haven't introduced unintended effects (ripple effects) – that proved features still work • Typically, vendors incrementally release features into independent test teams • This activity can provide confidence that key features are stable. The IT auditor will enquire about the level of retesting performed • Regression testing can be performed at various levels and may include functional and non-functional tests (e.g. reliability, usability, compatibility) • It often involves selecting, minimising and prioritising test cases.
Non Functional	This examines aspects like performance, usability and reliability at all test levels. Key types include: • **Performance Testing:** Checks if the system meets performance requirements (e.g. capacity, response time) • **Load Testing**: Assesses behaviour under load to detect issues like 'deadlocks' or 'memory leaks' (Data integrity faults) • **Stress Testing:** Pushes the system beyond its limits to identify failures • **Failover Testing:** Verifies the system's ability to handle failures and continue operations • **Reliability Testing:** Assesses reliability through fault detection and statistical models • **Compatibility Testing:** Ensures compatibility with different hardware, software or versions • **Scalability Testing:** Tests the system's ability to handle increased load or data volume • **Elasticity Testing:** Evaluates cloud or distributed systems' ability to scale resources dynamically • **Infrastructure Testing:** Validates infrastructure components for performance and uptime.
Security	• Challenges the SUT from external assessment and attacks, to assess the confidentiality, integrity and availability of the system and its data • Typically includes assessing against misuse and abuse of the software or system, often involving negative testing • Advanced security testing will involve defensive coding standards and defensive design focus.
Interface and API	• Verifies the correct exchange of data and control between components • Application Programming Interface (API) testing simulates end-user applications by generating API call parameters, setting environmental conditions and defining internal data affecting the API.
Configuration	• Configuration testing verifies the software's functionality under different specified configurations to ensure it meets the needs of various users.
Usability and Human-Computer Interaction	• Evaluates how easily end-users can learn to use the software, including testing software functions, supporting documentation and error recovery features.
Backup/Recovery	• Determines if, in the event of failure, a SUT item can be restored from backup to its pre-failure state • Backup/recover testing then focusses on testing the correctness of the test item's backup and the correctness of the restored state of the test item against its pre-failure state • Backup/recover testing can also be used to verify whether the backup and recovery procedures for the test item achieve specified recovery objectives • This type of testing may be carried out as part of a disaster recovery test.

References [3, 4, 5]

The test objective is important as it determines what the testing activity goal is. More aspects of the SUT will be challenged when the test objectives are varied. Quite often the IT auditor will examine a defect and discuss with the vendor the root cause and associated testing objective that may have been leveraged to detect the defect within the SLC.

IT AUDITOR CHECK

Ask the vendor about their test strategy and establish the scope of test objectives. The vendor's scope is an indicator that the SUT will be more robust.

If the focus is on functional requirements, then it may indicate that other important objectives have not been challenged, resulting in a higher risk of latent defects in the system (lower software product quality).

The test strategy will indicate the test phases and associated testing objectives so that the software product quality is as good as it can be, given the maturity of the SLC and requirement risks. The test technique defines the methods used to attain the test objective.

TEST TECHNIQUES

It is acknowledged that it is generally unfeasible to test everything. As a result a test strategy will define test activities (techniques) that will challenge the SUT as effectively and efficiently as possible. The selection of techniques is a test design activity, where the most efficient techniques are chosen for the specific objective:

- Various testing techniques exist that aim to improve the SUT's quality by generating test suites to detect as many failures as possible
- Testing techniques are used to maximise time available, can be used to target defect types and indicate the maturity of an organisation. They are used to detect and remove defects before the software is released
- Testing techniques can be categorised by the degree of information available about the SUT. Specification-based (black-box) techniques rely solely on input/output behaviour, while structure-based (white-box) techniques use internal design or code.

The following section highlights some of the more commonly used testing techniques for vendor discussions. Assessing the scope of testing techniques in use will provide an indicator on the organisation's testing effectiveness.

WHITE BOX TECHNIQUES

See Figure 2. White box testing involves building tests based on the structure of the code scaffolding inside a component. Typically this focuses on unit testing.

Unit testing includes static testing (e.g. code reviews), dynamic testing (e.g. statement/branch/path coverage) and complexity analysis (e.g. cyclomatic complexity).

As this type of testing requires programming skills (which is beyond the scope of this article) the following provides a high level description of the some of the unit testing techniques that the auditor can ask about during the vendor audit.

The highlighted examples in Table 2 are the techniques that generate real value added from a software engineering perspective (defect detection, software product quality).

IT AUDITOR HINT

Even if the auditor doesn't understand code, ask for a walkthrough of an example. Quite often it is found that the vendor may struggle to find an example or the organisation has defined Unit Testing as a System or User Acceptance level test. If so, bear in mind that the omission of a test phase may result in a lower software product quality level.

FIGURE 2. WHITE BOX TECHNIQUES

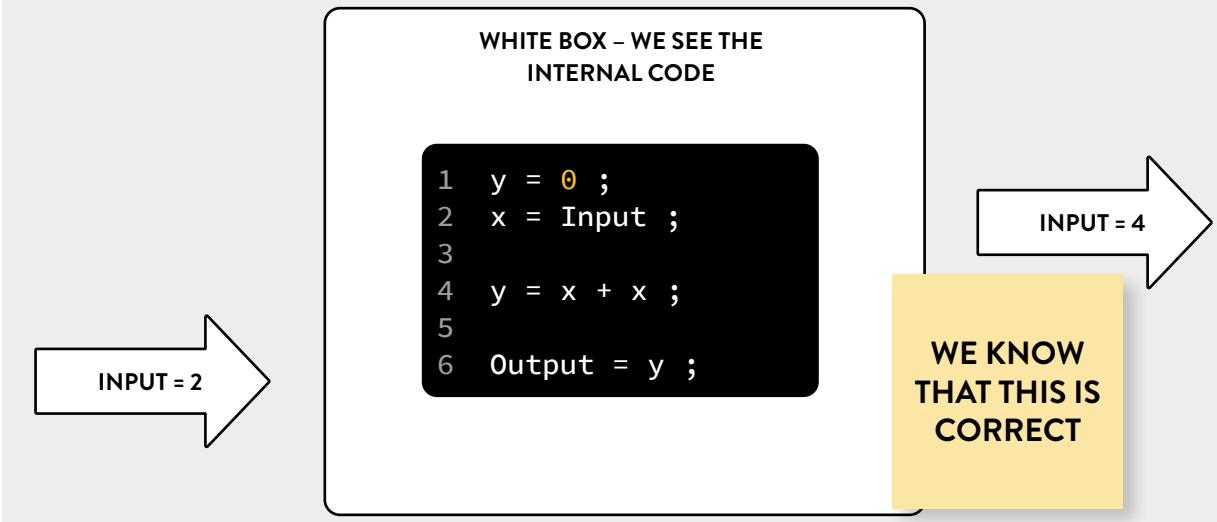


TABLE 2. TESTING TECHNIQUES

TESTING TECHNIQUE	DESCRIPTION	SCOPE
Statement Coverage	Ensures that every executable statement in the code is tested at least once.	Helps identify unexecuted code but may miss logical errors that depend on conditions.
Branch Coverage (Decision Coverage)	Tests all possible branches (true/false outcomes) of conditional statements (e.g. if-else).	More effective than statement coverage, as it checks both outcomes of each decision point.
Condition Coverage (Predicate Coverage)	Ensures that each Boolean sub-condition within a decision statement is evaluated both true and false at least once.	Provides better granularity than branch coverage but does not test all combinations of conditions.
Multiple Condition Coverage	Tests all possible combinations of Boolean conditions within a decision statement.	Offers the highest level of logic testing but requires more test cases.
Path Coverage	Ensures that every possible path through the code is executed at least once.	Comprehensive but impractical for large programs due to exponential growth in test cases.
Loop Testing	Focuses on testing loops with different execution scenarios (zero iterations, one iteration, many iterations, boundary values).	Essential for detecting infinite loops and incorrect loop conditions.
Data Flow Testing	Tracks variable definitions and their usage throughout the program to detect uninitialised variables unused variables or incorrect data flow.	Helps identify runtime errors and memory leaks but requires deep code analysis.
Control Flow Testing	Analyses the logical control paths within the program, ensuring that all possible execution flows are tested.	Helps uncover logical errors and unintentional dead code.
Mutation Testing	Introduces small modifications (mutants) in the code to check whether test cases can detect the changes.	Highly effective for assessing test suite quality but computationally expensive.

References [3,4,5,6]

BLACK BOX TECHNIQUES

This strategy relies on the requirements and specifications for deriving test scenarios rather than visibility of the underlying software code statements. No knowledge of the inner workings of the software code statements is required. Hence it is applicable for post unit testing phases by independent test resources.

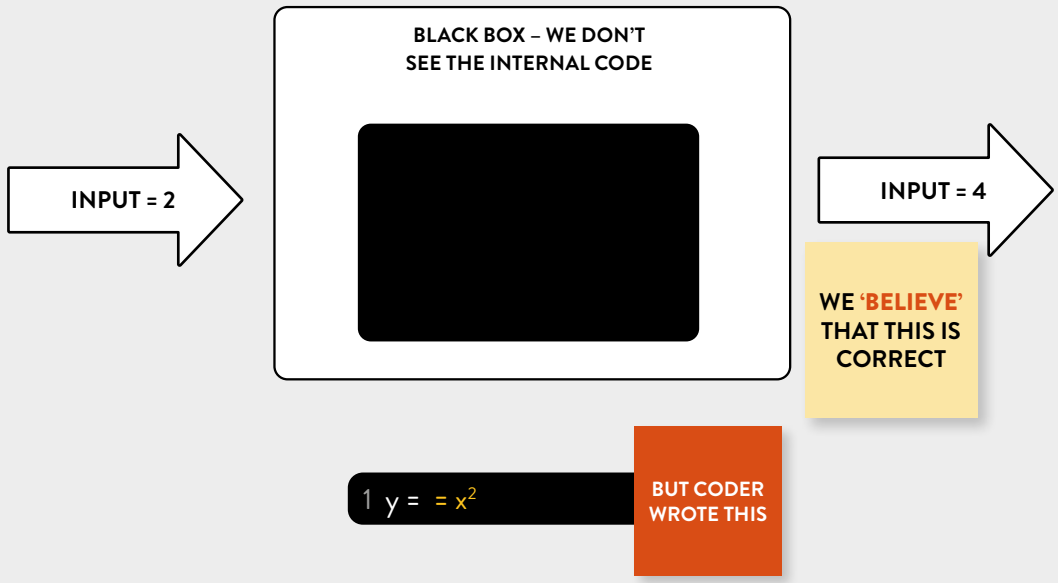
Because there is no visibility of the inner workings, an assumption is often made that the inner workings of the software code is verified. A good test strategy will treat this assumption as a risk and mitigate by applying more than one test objective and technique to a feature.

Efficient black box testing focuses on choosing a subset of tests that are efficient and effective at finding defects. This is a more effective approach than a randomly selected number of tests. An indicator of a vendor approach is where the vendor process takes time to assess, select, estimate and schedule the application of testing techniques. This approach of going slow (design) before going fast (execution) is often an indicator of good practice.

The following pages provide details on some of the common black box techniques.

‘An indicator of a vendor approach is where the vendor process takes time to assess, select, estimate and schedule the application of testing techniques.’

FIGURE 3. BLACK BOX TECHNIQUES



INSPECTION AND REVIEWS

The purpose of this technique is to find, early in the SLC, problems that may cause defects later.

There are several review types that are available:

1. One Third Presentation – Early feedback on technical content and solutions before completion. Support on the job training and reduces scope of rework effort.

2. Informal Review – Quick, email-based feedback for minor changes. Comments are reviewed via email for the author's resolution.

3. Code Walkthrough – Collaborative, real-time code review where the author explains the code to reviewers for defects and improvements. Focused on consistency and ease of maintainability.

4. Formal Review/Inspection – Structured, role-based review focusing on defect logging with solutions identified post review meeting. Very strong defect detection and assurance. Reviewers sign off on resolved issues.

Typically the following are suitable:

Requirements, design, plans, test cases, source code, user documentation and training material.

Formal review/inspections merit consideration as they are about twice as efficient as most forms of dynamic testing techniques (Caper Jones).

Review/inspection meetings should be small and include people who are independent of the artefact under review. The artefacts are made in advance of the review meeting. Questions and issues are recorded during the meeting for remediation post meeting. A record of the volume and type of issues are recorded. The artefact is reworked until acceptance.

Many vendors will not consider a key artefact complete (such as requirements and design) until it has been through a review process.

Review Date	09 May 2005
Review Type	Formal
Review Location	Dev Mtg Room
Action List Produced	Yes

Review Preparation Time (hr)	2.5
Review Meeting Time (hr)	1
Follow up Review Required (Yes/No)	No
Review Follow Up Time (hr)	N/A

Issues Raised	**Base**	**New**	**Total**
Ma	6	-	6
Mi	10	-	10
I	7	-	7
Q	2	-	2
Total	25	-	25

EQUIVALENCE PARTITIONING

Inputs are grouped into categories based on factors like expected results, program behaviour, or whether they are valid or invalid. One test case is then selected from each group to represent the whole category[5].

Only one value needs to be tested within each range or equivalence class:

- If one test case in an equivalence class detects a defect, all other test cases in the same equivalence class are likely to detect the same defect
- Similarly of one test case in the equivalence class detects no defect.

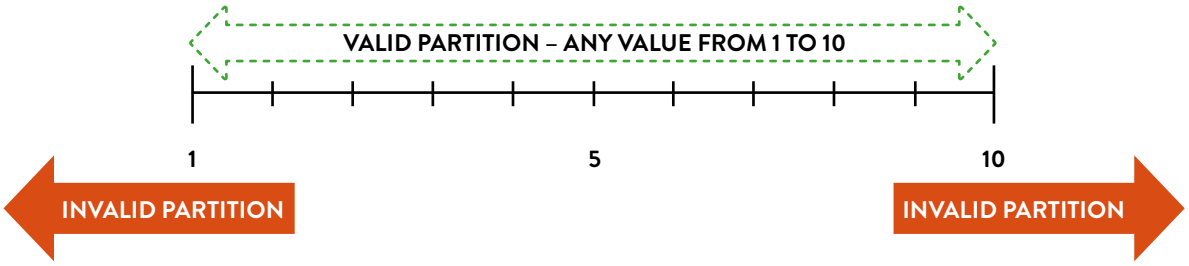
Used to reduce the number of test cases.

Suited to systems where much of the input data takes on values within sets.

Any data within a class is equivalent (in test terms) to another any other value within the same class.

Used to determine:

- Functional suitability (completeness and correctness)
- Usability (user error protection)
- Reliability (availability)
- Security (confidentiality, integrity, non-repudiation, accountability, authenticity).



BOUNDARY VALUE ANALYSIS

Equivalence Partitioning testing naturally leads onto Boundary Value Analysis (BVA).

BVA adds focus onto the boundaries between equivalence classes, simply because this is where so many defects reside[5,6].

For a partition defined as integers from one to 10 inclusive, there are two boundaries, where the lower boundary is one and the upper boundary is 10 and these are the test conditions. Robustness testing extends this by including out-of-range values to check error handling by selecting the next value beyond the boundary value (0 and 11)[4].

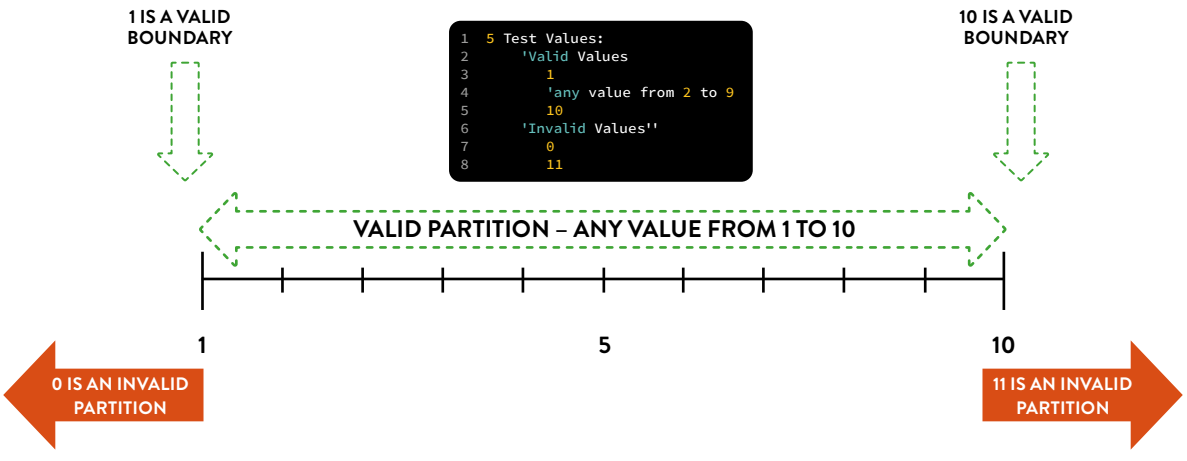
Can significantly reduce the number of test cases that must be created.

Suited to systems where input values take on values which reside within ranges or sets.

Applicable to all test phases.

Used to determine:

- Functional suitability (completeness and correctness)
- Performance (time behaviour, capacity)
- Usability (user error protection)
- Reliability (fault tolerance)
- Security (data integrity).



DECISION TABLE TESTING

Represents conditions (inputs) and actions (outputs) in a table, systematically deriving test cases for all possible condition-action pairs.

Decision table testing[5, 6] uses a model of the logical relationships (decision rules) between conditions (inputs) and actions (outputs) for the test item in the form of a decision table:

- Each action (output) is the expected outcome(s) for the test item
- A set of decision rules defines the required relationships between conditions and actions.

Each decision rule, which defines the relationship between a unique combination of the test item's conditions and actions, is a test coverage item.

Used to capture complex business rules and help in test case identification.

Conditions represent input conditions.

Actions are logic that should be executed dependent on the combinations of input conditions.

Each rule defines a unique combination of input that will execute the associate action.

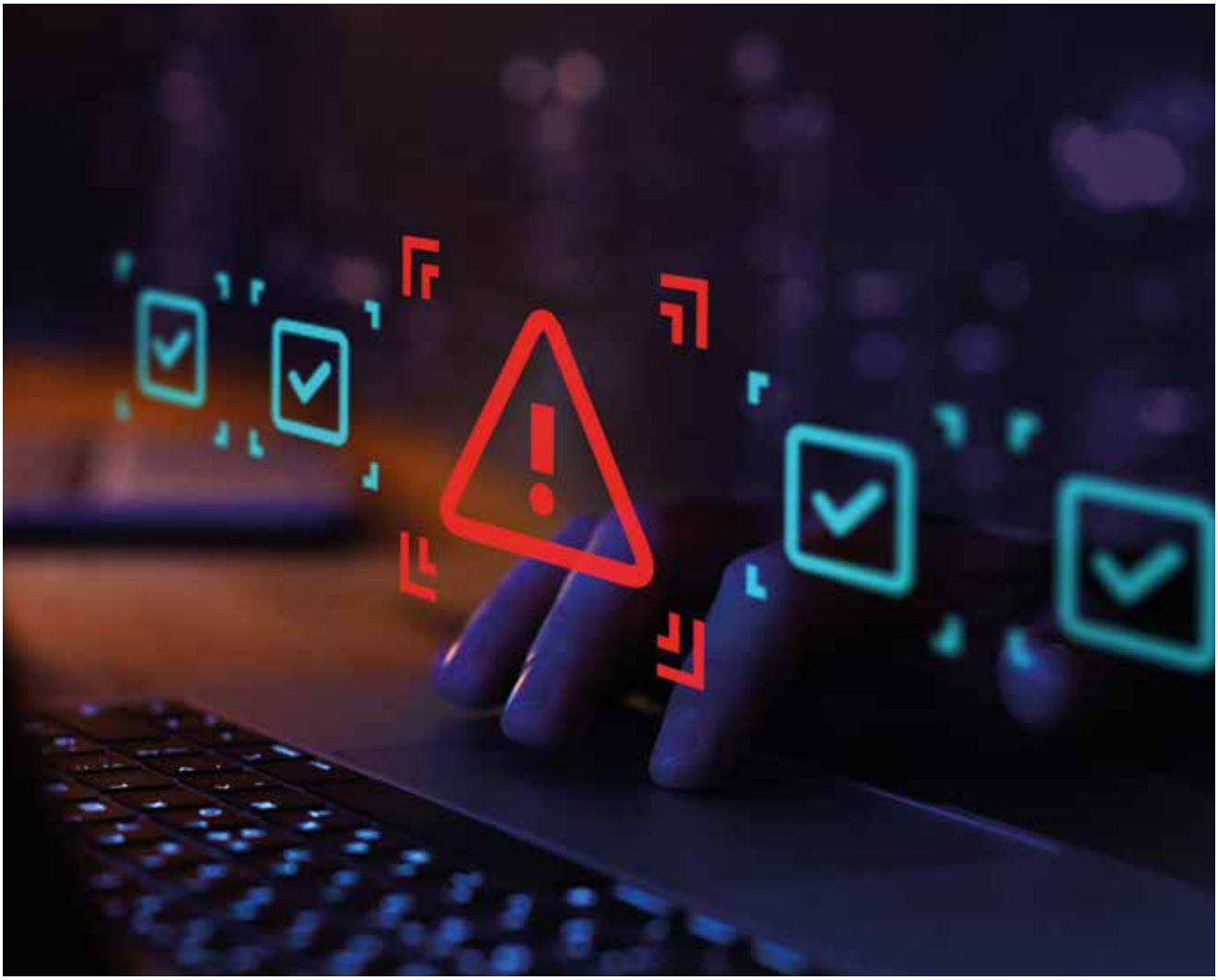
Testing:

Create one test for each rule. Apply equivalence classes where necessary.

Used to determine:

- Functional suitability (completeness and correctness)
- Compatibility (coexistence)
- Performance (time behaviour)
- Usability (user error protection).

Condition (input)	Rule 1	Rule 2	Rule 3	Rule 4
Wage Earned	Y	N	Y	N
End of Pay Period	Y	Y	N	N
Action (output) Pay Tax	Y	N	N	N



STATE TRANSITION TESTING

State transition testing[4,7] checks how a system (can only) move between different states based on inputs and events.

It treats the system as a 'finite-state machine' and creates test cases to ensure all states and transitions are covered.

A state is a static condition and a transition is a command that moves it from one static state to another static state.

These transitions can be shown using diagrams or tables.

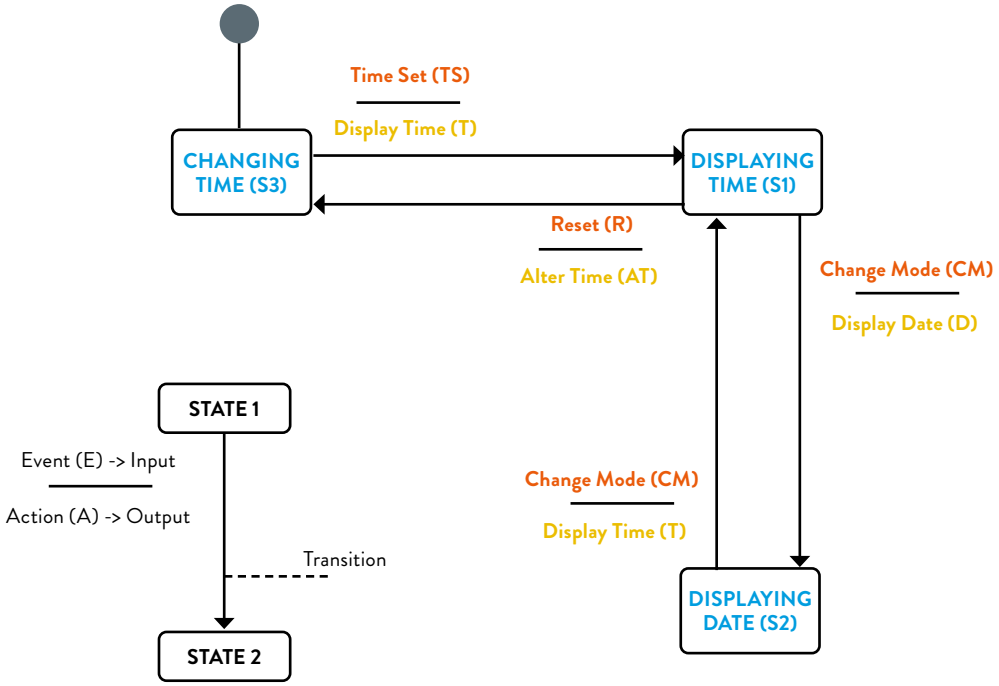
Effective system design technique that can be leveraged for test creation.

Useful for directing test effort by identifying states, events, actions and transitions.

State diagrams are easier to comprehend.

State transition tables are easier for use in a systematic manner. Tables are especially useful for identifying both valid and invalid transitions, which helps in testing high-risk systems.

Not applicable where the system has no state change.



CURRENT STATE	EVENT	ACTION	NEXT STATE
Displaying Time (S1)	Change Mode (CM)	Display Date (D)	Displaying Date (S2)
Displaying Time (S1)	Reset (R)	Alter Time (AT)	Changing Time (S3)
Displaying Date (S2)	Change Mode (CM)	Display Time (T)	Displaying Time (S1)
Changing Time (S3)	Time Set (TS)	Display Time (T)	Displaying Time (S1)

Other techniques for the reader to consider:

- **Pair-Wise Testing:** Test for every pair of input values. (This is a highly successful approach to detecting defects and reducing the number of test cases and an indicator of a good quality focused organisation)
- **Smoke Testing:** Ensures newly released features will work before in-depth testing begins. The name originates from early electronics and hardware testing where engineers would power on a new electrical circuit for the first time. If it starts to smoke then something is wrong. If the smoke test fails in software engineering, then the release is not stable for further testing
- **Use Cases[8]:** Used to exercise a system's functionalities from the start to finish by testing each of its individual transactions. Uses cases (Ivar Jacobson) define scenarios that describe the use of a system by an 'actor'. An actor can be a user or another system. A scenario can be a sequence of steps between the actor and the system
- **Error Guessing[5,6]:** Involves the design of a checklist of defect types that may exist in the test item, allowing the tester to identify inputs to the test item that may cause failures, if those defects exist in the test item. Test cases are created based on known fault patterns, past defects and the tester's expertise
- **Exploratory Testing:** Combines learning, test design and execution in real time, allowing testers to adapt dynamically based on observations, system peculiarities and risk factors. Software house may assign 5% of the test execution schedule to exploratory testing to allow for a more comprehensive assessment of the SUT
- **Data Flow Testing:** Analyses variable (data) definitions, usage and lifetimes within the control flow. Powerful tool to detect errors in data transfer, initialisation and processing within code.

Section References[3,4,5,6,7,8,9,10,12,13]

'Software house may assign 5% of the test execution schedule to exploratory testing to allow for a more comprehensive assessment of the SUT.'

SUMMARY

The following provides some considerations when looking at a vendor's test strategy.

ASPECT	GOOD STRATEGY	POOR STRATEGY
Clarity and Structure	Clearly defined goals, scope and approach. Project budget and time allocated for test design.	Vague or lacks structure, making it difficult to follow.
Coverage	Covers all critical aspects (functional, performance, security, etc.).	Misses key testing areas or lacks depth in coverage. Likely to result in more defects in operational use.
Test Objectives	Clearly aligned with project requirements and risks.	Unclear objectives or misaligned with business needs.
Flexibility	Adapts to changes in requirements, scope or risks.	Rigid, failing to accommodate project evolution.
Risk Management	Identifies, prioritises and mitigates risks.	Ignores risks or lacks a plan for handling failures. Coverage of testing may be misaligned to feature acceptance.
Test Techniques	(SOP/Wiki/Work Instruction) Uses a mix of appropriate techniques (e.g. exploratory, automated, boundary analysis).	Relies on a single technique, leading to gaps in defect detection.
Tools and Resources	Selects and utilises proper testing tools effectively.	Uses inefficient or irrelevant tools or lacks tool support.
Test Data Management	Well-planned, diverse test data for realistic scenarios.	Poor or unrealistic test data, leading to ineffective testing.
Defect Tracking and Reporting	Well-defined defect tracking process with detailed reports.	Unstructured defect reporting, making debugging difficult. Ineffective test phase reporting making it difficult to assess the quality of the SUT at that time.
Communication and Collaboration	Clear coordination among developers, testers and stakeholders.	Lack of communication, causing misunderstandings and delays.
Efficiency and Execution	Well-planned test execution with prioritisation of critical tests.	Disorganised execution, wasting time on low-priority tests.

Reference[3]

'Formal inspections are almost twice as efficient as dynamic testing in detecting defects.'

Test techniques are designed to detect defects pertaining to specific scenarios, technologies and within the various layers involved in building software. A good vendor's testing strategy will seek to identify techniques to seek defects and then confirm feature correctness.

Defect detection efficiency relates to how well a test technique can find defects so they can be corrected.

Individual dynamic test techniques have relatively low defect detection efficiency (given their targeted defect focus). This is why an aggregation of techniques are needed. Formal inspections are almost twice as efficient as dynamic testing in detecting defects. Their use can be an indicator of medium to high quality software vendors as testing alone is not enough [2,11]. The measurement of testing activities will be discussed in a subsequent article.

Take home point for the next vendor audit: the more varied the techniques applied, the more likely that defects are detected for removal, leading to better software product quality before it arrives at the regulatory domain. As an auditor, getting an insight to the types of testing being conducted can provide an indicator on the quality maturity of the organisation that may not be so apparent in the associated documentation.

The next article will examine the management function of documentation and reporting on software product quality (rather than documentation quality).

REFERENCES

1. Software Testing: Measuring Vendor Software Quality – Part 1, Quasar #170
2. General Principles of Software Validation; Final Guidance for Industry and FDA Staff, FDA, 2002
3. Guide to the Software Engineering Body of Knowledge 4.0, IEEE Computer Society, 2024
4. IEEE 29119-4:2015 – Software and Systems Engineering — Software Testing — Part 4: Test Techniques. IEEE, 2015
5. BS 7925-2:1998 – Software Testing: Software Component Testing Standard. British Standards Institution, 1998
6. The Art of Software Testing, Myers, Wiley, 1979
7. A Practitioner's Guide to Software Test Design, Copeland, Artech House, 2004
8. Object-Oriented Software Engineering: A Use Case Driven Approach. Addison-Wesley, Jacobson et al,1992
9. Test Computer Software Kane, Cem, Falk, Nguyen (99)
10. Software testing Techniques, Beizer, Bosirs, Van Nostrand Reinhold, 1990
11. Applied Software Measurement, Global Analysis of Productivity and Quality, Caper Jones, McGraw Hill, 2008)
12. The Art of Electronics, Horowitz et al, Cambridge University Press, 1989
13. Rapid Development. Taming Wild Software Schedules, McConnell, Microsoft Press, 1996.

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.