



Barry McManus



Hugh O'Neill

# AUTOMATION TESTING

---

In this series of articles, the authors have briefly looked at how a combination of testing practices can be used to provide information on the software product quality and, by association, the QMS (Quasar issue 170 to 172).

Seeking out defects is expensive and software testing needs to be efficient by performing tests as fast and economically as possible (Quasar 171). Test automation can reduce the effort required to run tests and can significantly increase the volume of testing within a set timeframe. The IT Auditor is frequently faced with test automation during an audit. This article introduces test automation, what it can and can't do and provides some considerations for the next IT audit.

*Note: this text does not include software tools that are used to manage manual test activities.*

### THE NEED FOR TEST AUTOMATION

Manual testing is a laborious and time-consuming process. This is a common complaint against UAT validation. The FDA's CSA draft guidance<sup>1</sup> discussed at length test case structure to counter this. Automation is a component of the testing strategy (Quasar 171), that when conducted correctly, will help manage and reduce test execution effort whilst widening the defect detection net.

**Definition:** Test automation is the process of writing a computer program (script) to do testing that would otherwise need to be done manually. Tools and scripts perform test execution, manage test data and analyse results without continuous human intervention. It complements manual testing by taking over repetitive, predictable tasks, freeing human testers for cognitive thinking and higher-level validation<sup>2,3</sup>. It is more expensive to create and maintain than manual tests, but cheaper to execute in comparison<sup>4</sup>.

**TABLE 1. SIMPLE EXAMPLE OF ONE AUTOMATION TEST SCRIPT GENERATING MULTIPLE TEST CASES FOR VALID LOGIN SCENARIOS**

**Simple Cost Benefit Analysis** (test execution only) of a real world system test scenario of 8,500 system tests: -35 testers to conduct regression system testing of application versus costs for the corresponding test automation approach.

MANUAL TESTING	AUTOMATION TESTING
<b>Manual test execution (results and reporting):</b> 20 minutes/test # of tests: 8,500 Total test time (1 tester): 170,000 mins	<b>Test automation execution (results and reporting):</b> ½ minute/test # of tests: 8,500 Total test time (1 auto PC): 4,250 mins
<b>Time for 35 testers across 35 PCs:</b> 4857.14 mins ~ 81 hours	<b>Time for 1 automation tester overseeing 35 PCs:</b> 121.42 mins ~ 2hrs
Employee costs (IT Jobs Watch software Engineer) ~26/hr Employee cost for 81 hrs: £6,561.00	Employee costs (IT Jobs Watch software Engineer) ~26/hr Employee cost for 2hrs: £52.00
<b>MANUAL EXECUTION COSTS: £6,561.00</b>	<b>AUTOMATION EXECUTION COSTS: £52.00</b>

This cost benefit analysis focuses on execution savings only.

```
@login-validation @regression
Scenario Outline: Login validation with different credentials
Given I have username '<username>' and password '<password>'
When I attempt to login
Then I should see '<expected_result>'
And the login status should be '<status>'
```

Examples: Valid Login Scenarios

Test_case_ID	username	password	expected_result	status	Description
Login_001_valid	john@example.com	correct123	Welcome message	success	Correct username and password
Login_002_ValidCase	JOHN@EXAMPLE.COM	correct123	Welcome message	success	Uppercase email validation
Login_003_ValidTrim	john@example.com	correct123	Welcome message	success	Username with trailing spaces
Login_007_EmptyUser		correct123	Username required	failed	Empty username field
Login_008_EmptyPass	john@example.com		Password required	failed	Empty password field
Login_009_BothEmpty			Credentials required	failed	Both fields empty

Once tests have been automated, they can be run quickly and repeatedly. This is often the most cost effective method for software products that have a long maintenance life, because even minor patches can cause features to break which were previously working<sup>5</sup>.

The auditor role is to discern the quality of the testing and the quality of the automation. To do this, the auditor needs to ascertain the quality of the information presented.

## BENEFITS OF MANUAL VS AUTOMATION TESTING

### MANUAL TESTING

- a) **Critical Thinking:** Essential for exploratory testing, usability and scenarios requiring human judgment<sup>6</sup>.
- b) It is **flexible** as testers can quickly **adapt** to unexpected results.

- c) **Lower initial investment:** minimal upfront costs in tools and specialised skills.
- d) Valuable in early development phases for information feedback, when features are unstable.
- e) Critical for ad-hoc and user experience verification.

### AUTOMATION TESTING

- f) **Repeatability:** Provides repeatability for regression and smoke tests<sup>7,8</sup>.
- g) **Speed:** It allows more tests to be executed<sup>4</sup>.
- h) **Error reduction:** Reduces human error in repetitive execution.
- i) **Parallelism:** Enables parallel testing across platforms and browsers – increases breadth of testing.

- j) **Technical testing:** It allows for technical tests, such as load testing, that are unfeasible for a manual approach<sup>4</sup>.
- k) **Continuous runs:** Facilitates continuous execution 24/7.
- l) **Continuous builds:** Supports continuous integration/delivery (CI/CD) pipelines.
- m) **Coverage:** Increases test coverage by running large data sets or stress tests<sup>9</sup> Builds reusable test libraries that scale with the system under test.
- n) **Expansion:** Allows for the growth of test suites without proportional manual effort.

As part of an end-to-end testing strategy, automation frees up manual testers to focus on new features, usability, exploratory and risk focused testing. This leverages the real benefit of manual testers, which is critical thinking.

TABLE 2. TESTING ACTIVITY, DEFECTS TARGETED AND SUITABILITY TO TEST AUTOMATION

AUTOMATED TEST TYPE	DEFECTS TARGETED	AUTOMATION SUITABILITY
<p><b>Unit Testing:</b> focuses on testing individual software components or modules in isolation to validate that each unit of code performs as designed.</p> <p>Small, repeatable, stable functions; executed with every build<sup>2</sup>.</p> <p>Ensures that a data item is the correct data item, that it is processed correctly within a small component and that its error handling is managed.</p> <p>Facilitates code refactoring.</p>	<p>Logic errors in individual functions and methods.</p> <p>Incorrect calculations or data transformations.</p> <p>Edge case handling failures within single components.</p> <p>Invalid input processing at the unit level.</p> <p>Return value errors and boundary condition failure.</p>	<p><b>High</b></p> <p><b>Why:</b> Unit tests provide fast feedback, are predictable, consistent and reproducible. They have minimal external dependencies, making them ideal for automated execution in development workflows.</p> <p><b>Constraints:</b> Requires well-structured, modular code architecture. It may require complex test harnesses and may omit integration issues between components.</p>
<pre> public class Calculator {     public string CheckNumber(int number)     {         if (number &gt; 0)         {             return 'Positive';         }         else if (number &lt; 0)         {             return 'Negative';         }         else         {             return 'Zero';         }     } }  [Test] public void CheckNumber_ReturnsPositive_WhenNumberGreaterThanZero() {     var result = _calculator.CheckNumber(5);     Assert.AreEqual ('Positive', result); }  [Test] public void CheckNumber_ReturnsNegative_WhenNumberLessThanZero() {     var result = _calculator.CheckNumber(-3);     Assert.AreEqual ('Negative', result); }  [Test] public void CheckNumber_ReturnsZero_WhenNumberEqualsZero() {     var result = _calculator.CheckNumber(0);     Assert.AreEqual ('Zero', result); }                     </pre>		
<p>The unit test phase is complemented by the use of code analysers to consider complexity, standard adherence, coverage and memory checks.</p>		

AUTOMATED TEST TYPE	DEFECTS TARGETED	AUTOMATION SUITABILITY
<p><b>Smoke Testing:</b> Quick checks for installation health that can run automatically after deployment<sup>7,8</sup>.</p> <p>It is verification to ensure critical features work after deployment or major changes and to prevent delay with concurrent testing in an environment.</p> <p>Along with unit tests, this is usually the first type of automation testing that is created.</p>	<p>Build deployment failures and configuration errors.</p> <p>Critical path breakages in core functionality.</p> <p>Basic functionality regressions after code changes.</p> <p>Major integration failures between system components.</p> <p>Environment setup and configuration issues.</p>	<p><b>High</b></p> <p><b>Why:</b> Smoke tests represent a minimum set of critical tests that can be executed quickly and consistently in CI/CD pipelines and any deployment approach.</p> <p><b>Constraints:</b> Limited test coverage scope, may miss deeper functional issues, requires careful selection of truly critical test cases. Not suitable to verify the functionality of software product.</p>
<p><b>Integration Testing:</b> Tests the interactions between integrated components.<sup>10</sup></p> <p>Ensures modules work together; automation speeds up repeated checks<sup>2</sup>.</p> <p>Tests the transfer of data items between components.</p>	<p>Interface mismatches between software components.</p> <p>Data format inconsistencies across module boundaries.</p> <p>Communication protocol errors between services.</p> <p>Incorrect API contracts and interface specifications.</p> <p>Timing and synchronisation issues in component interactions.</p>	<p><b>High</b></p> <p><b>Why:</b> Automation Testing facilitates the integration test that is conducted below the GUI screens.</p> <p><b>Constraints:</b> Environment setup, external dependency management, data synchronisation challenges, longer execution times than unit tests.</p>
<p><b>Application Programming Interface (API) Testing:</b> Tests an API to verify correct data exchange, functionality, reliability and performance between software components. For example, a web service.</p> <p>If APIs have stable interfaces; automated tools can validate responses quickly<sup>7</sup>.</p>	<p>Security vulnerabilities in API endpoints.</p> <p>Compatibility problems preventing seamless client interaction.</p> <p>Incorrect HTTP status codes and response formats.</p> <p>Malformed request/response payloads.</p> <p>Authentication and authorisation failures.</p>	<p><b>Very High</b></p> <p><b>Why:</b> APIs provide structured, predictable interfaces that are well-suited to automated testing with consistent input/output validation.</p> <p><b>Constraints:</b> Requires sophisticated test data management, environment synchronisation and API versioning compatibility handling</p>
<pre> gherkin Feature: User API  Scenario: Create new user via API   Given I have a valid API key   When I send a POST request to "/api/users" with     """     {       "name": "John Does",       "email": "john@example.com"     }     """   Then the response status should be 201   And the response should contain a user ID   And the user should exist in the database           </pre>		
<p><b>Graphical User Interface (GUI) Testing:</b> Tests the graphical user interface to ensure proper functionality, usability and visual correctness of user-facing elements.</p> <p>Verifies that the correct action occurs when an element is selected.</p>	<p>Layout and rendering issues across different browsers and devices.</p> <p>Button/link functionality failures and navigation problems.</p> <p>Form validation problems and user input handling errors.</p> <p>Cross-browser compatibility issues.</p> <p>Accessibility compliance failures.</p>	<p><b>Moderate</b></p> <p><b>Why:</b> GUI automation can consistently execute repetitive test scenarios from the application front end.</p> <p><b>Constraints:</b> It is reliant to the design approach of the GUI. Tests can be brittle due to frequent GUI changes, complex element identification across browsers results in a high maintenance overhead for test scripts.</p>
<p><b>Regression Testing:</b> Regression testing re-tests previously working functionality to ensure new changes haven't introduced defects in existing features<sup>11</sup>. Vital for execution after smoke testing.</p> <p>Every build/release should feature regression test runs.</p>	<p>Previously fixed bugs reappearing after code modifications.</p> <p>New feature impact on existing functionality.</p> <p>Code refactoring side effects on established features.</p> <p>Configuration change impacts on system behaviour.</p> <p>Unintended behavioural changes in stable components.</p>	<p><b>Very High</b></p> <p><b>Most suitable:</b> repetitive and stable test cases across all releases<sup>9</sup>.</p> <p><b>Why:</b> The repetitive nature of regression testing makes it ideal for automation, allowing consistent execution of large test suites.</p> <p><b>Constraints:</b> High test maintenance overhead, lengthy execution times for comprehensive suites, complex test data management requirements.</p>

TABLE 2. TESTING ACTIVITY, DEFECTS TARGETED AND SUITABILITY TO TEST AUTOMATION (CONT.)

AUTOMATED TEST TYPE	DEFECTS TARGETED	AUTOMATION SUITABILITY
<p><b>Performance Testing:</b> Evaluates system behaviour under various load conditions, across various geographical locations, measuring response times, throughput and resource utilisation.</p> <p>Requires automated tools to simulate load/stress<sup>8</sup>.</p> <p>Highly skilled activity.</p> <p>The following three automation test types (A-C) are specific performance tests.</p>	<p>Slow response times under normal and peak loads.</p> <p>Memory leaks and resource consumption issues (software degradation).</p> <p>Database and network bottlenecks.</p> <p>Poor scalability characteristics.</p> <p>Inefficient algorithms causing performance degradation.</p>	<p><b>High</b></p> <p><b>Why:</b> Requires precise load generation, consistent test conditions and measurable metrics that manual testing cannot reliably provide.</p> <p><b>Constraints:</b> Expensive infrastructure requirements, complex environment setup, sophisticated result analysis and interpretation needs.</p>
<pre> Pid 15506 is ( )  Time      Idle CPU   Free   System Mem          PID 1 13:32:36  0.0%(B[m[39;49msi, (B[m[39;49(B[m 23791628k 13:31:37  0.0%(B[m[39;49msi, (B[m[39;49(B[m 23790140k 13:31:39  0.0%(B[m[39;49msi, (B[m[39;49(B[m 23789080k 13:31:40  0.0%(B[m[39;49msi, (B[m[39;49(B[m 23788160k 13:31:42  0.0%(B[m[39;49msi, (B[m[39;49(B[m 23788092k 13:31:44  0.0%(B[m[39;49msi, (B[m[39;49(B[m 23787416k 13:31:45  0.0%(B[m[39;49msi, (B[m[39;49(B[m 23787308k                     </pre>		
<p><b>A. Load Testing:</b> Tests system behaviour under expected normal and peak load conditions to verify that performance requirements are met under realistic usage scenarios.</p>	<p>Performance degradation under expected load levels.</p> <p>Database connection pool exhaustion.</p> <p>Server resource limitations and capacity issues.</p> <p>Network bandwidth bottlenecks.</p> <p>Session management and concurrency problems.</p>	<p><b>High</b></p> <p><b>Why:</b> Load testing requires precise, repeatable load generation and consistent measurement conditions that are impractical to achieve manually.</p> <p><b>Constraints:</b> Expensive test environment requirements, realistic data generation needs, complex result analysis and performance baseline establishment.</p>
<p><b>B. (Data) Volume Testing:</b> Tests system behaviour with large amounts of data to identify issues related to data processing, storage capacity and retrieval performance under high-volume conditions.</p>	<p>Database performance degradation with large datasets.</p> <p>Memory overflow and storage capacity errors.</p> <p>File system limitations and disk space issues.</p> <p>Search and query performance problems with large data volumes.</p> <p>GUI rendering issues and timeout problems with extensive data sets.</p>	<p><b>High</b></p> <p><b>Why:</b> Requires automated large-scale data generation, consistent test conditions and measurable performance outcomes that are impractical manually.</p> <p><b>Constraints:</b> Complex test data generation and management, significant storage infrastructure requirements, time-intensive execution and cleanup procedures.</p>
<p><b>C. Stress Testing:</b> Pushes systems beyond normal capacity limits to identify breaking points, failure modes and recovery capabilities.</p>	<p>System crashes and failures under extreme load conditions.</p> <p>Resource exhaustion failures (memory, CPU, disk).</p> <p>Poor error handling and recovery under stress conditions.</p> <p>Memory leaks and performance degradation under sustained load.</p> <p>Inadequate failover mechanisms and disaster recovery procedures.</p>	<p><b>High</b></p> <p><b>Why:</b> Requires controlled generation of extreme conditions and precise load patterns that are dangerous and impractical to perform manually.</p> <p><b>Constraints:</b> Risk of system damage requiring isolated environments, complex failure analysis procedures, expensive infrastructure and monitoring requirements.</p>
<p><b>Security Testing:</b> Identifies vulnerabilities, security flaws and potential threats in applications to ensure data protection and system integrity.</p>	<p>SQL injection and code injection vulnerabilities.</p> <p>Cross-site scripting (XSS) and cross-site request forgery.</p> <p>Authentication bypasses and authorisation failures.</p> <p>Data exposure and privacy violations.</p> <p>Cryptographic implementation weaknesses.</p>	<p><b>Partial</b></p> <p>Some scans (e.g. SQL injection, XSS) can be automated, but deep analysis requires manual expertise<sup>6</sup>.</p> <p><b>Why:</b> Automated security scanning can consistently check for known vulnerability patterns and security misconfigurations.</p> <p><b>Constraints:</b> Requires specialised security expertise and high IT skillset. It can generate false positives/negatives, limited coverage of complex attack scenarios, manual verification often required.</p>

## LIMITATIONS OF AUTOMATION TESTING

- **Software Lifecycle (SLC):** Test automation is development and it must follow a SLC process
- **Costs:** As there are more than just 'coding' activities, there is a high up front cost to build automation tests
- **Troublesome Software Under Test:** If the system is unstable or changing rapidly, automation testing may be inefficient<sup>12</sup>
- **Not a Human Replacement:** Test automation does not replace the need for manual testers. Human testers are still needed for exploratory, usability and domain knowledge validation<sup>6</sup>. Bear this in mind when auditing suppliers. Manual testers will find more defects than automation tests
- **Test Abandonment:** Poorly designed automated tests often get discarded due to high maintenance costs<sup>10</sup>
- **False Confidence:** Automation may pass scripts while missing untested edge cases or due to error in the automated code logic<sup>7</sup>. Just because an automated test case passes, it does not mean that there are no defects in the software

- **The Pesticide Paradox:** A test is most likely to find a new defect during the first time it is executed. The likelihood of finding new defects reduces, unless the code has changed or is affected by a change in a different part of the software or environment (ripple effect)<sup>7</sup>. Running the same tests repeatedly reduces effectiveness. Requires regular review and refresh of automated test suites<sup>4</sup>
- **Maintenance Overheads:** When the software changes, there is a follow-on impact on the automation tests. When it takes more effort to update the automation tests than the software, the automation is abandoned. Ease of maintainability is one of the key foundations for automation design
- **3rd Party Automation Tools:** Software sold by vendors and can be subject to defects
- **Effectiveness:** Automating a suite of tests does not make them any more effective than before. All the automation has done has improved the efficiency of the execution<sup>4</sup>. The effectiveness of any test case (automated/manual) comes in its design (Quasar 171).

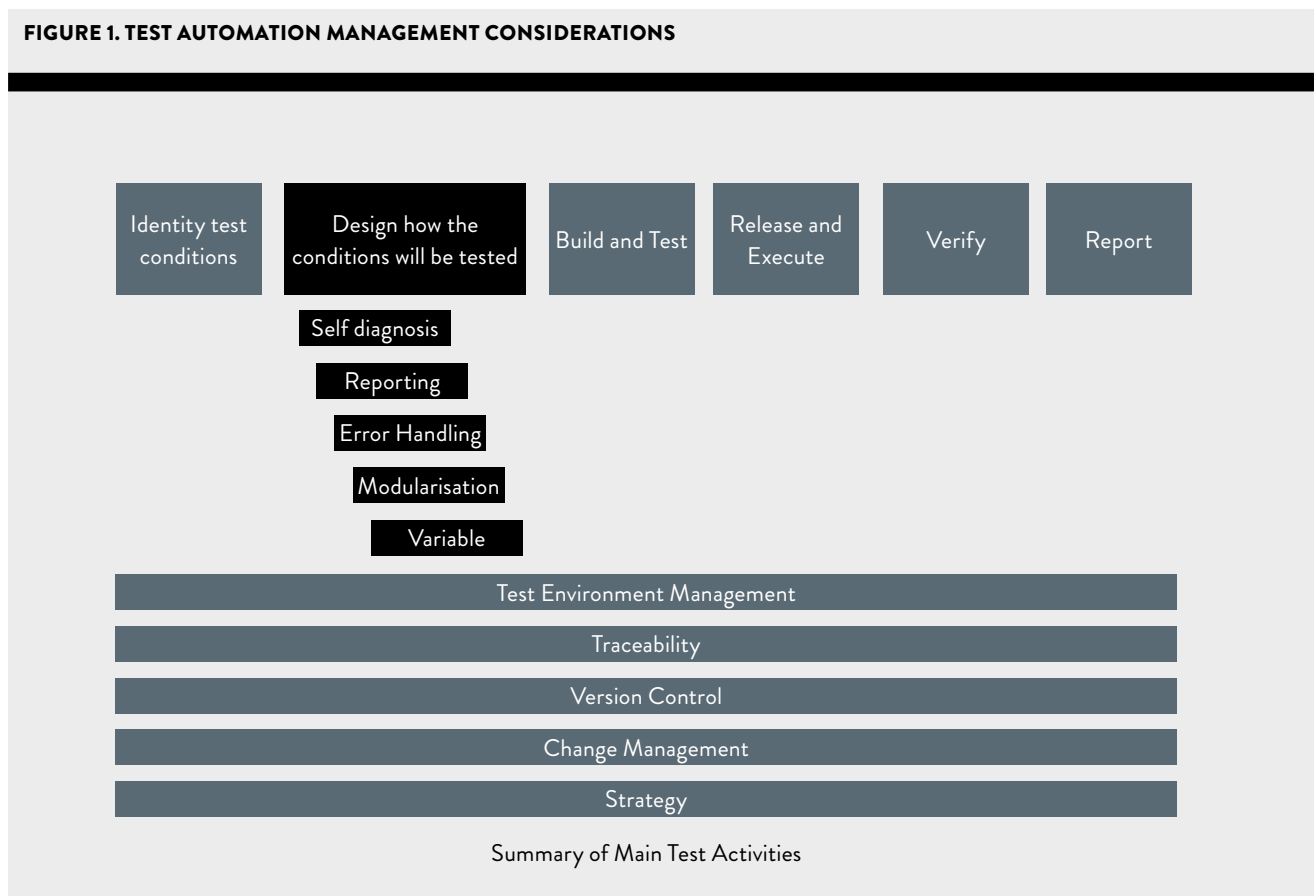
[Auditor hint] There is a greater reliance on the quality of the automated tests and the need to verify the correctness of their expected outcomes. Ask the auditee how this is managed.

## AUTOMATION TEST MANAGEMENT

Effective test automation takes a significant effort and requires good management to realise its benefits. The same practices required for managing a manual test approach applies, but with development considerations: strategy, planning, design, coding, verification, release, maintenance and reporting. (See figure 1.)

'When it takes more effort to update the automation tests than the software, the automation is abandoned. Ease of maintainability is one of the key foundations for automation design.'

FIGURE 1. TEST AUTOMATION MANAGEMENT CONSIDERATIONS



SOME KEY ASPECTS FOR THE IT AUDITOR TO CONSIDER:

**The Need to Follow a Software Lifecycle (SLC):** Testing automation is most effective when embedded into a defined software lifecycle (e.g. Waterfall, V-Model, Agile). It ensures traceability from requirements to design, to implementation, to testing and maintenance<sup>3</sup>. GxP and regulated industries require documented lifecycle controls to maintain compliance. Ensure that there is design and standards to ensure effective use and ease of maintainability.

**Strategy:** What to automate and when to automate are crucial decisions to make. Selecting the correct features of the product for automation largely determines the success of the automation. Automating unstable features or features that are undergoing changes should be avoided. Ask the auditee about their test strategy, what is automated, how and when. Ask them how they know that the strategy is effective and then to prove it.

**Need for Effective Test Run Information:**

Automated test reports should include execution logs, environment info, error details and trends. Integration with test management tools (e.g. JIRA, TestRail, Azure DevOps) is key<sup>4</sup>.

**Cleansing Automated Tests:** Check for the review and removal of redundant or ineffective tests that no longer add value. Maintain test traceability to updated requirements<sup>3</sup>.

**Explainability, Self-Diagnosis and Logging:** Automated tests should be self-diagnosing, report (and explain) results, log failures clearly and identify root causes. Logging and reporting typically capture test ID, environment, input data, expected vs actual results. Explainability provide context on breadth and depth of testing. Strong diagnostics reduce wasted time on false positives and enable faster debugging<sup>4</sup>.

**Tool and environment qualification:** Ensure that the tools in use are qualified to be fit for their intended needs.

‘GxP and regulated industries require documented lifecycle controls to maintain compliance. Ensure that there is design and standards to ensure effective use and ease of maintainability.’

FIGURE 2. EXAMPLES OF SOME TOOLS THAT ARE USED IN THE SOFTWARE LIFECYCLE

```

Requirement management tools:
** Capture, track, risk assesses, prioritize & trace requirements.
// IBM DOORS, Helix RM, Jira (with pugins).

Design tools:
** Tools used to design logical components & data flows.
//Design: Auto design generators, UML, Enterprise architect, Rational Rhapsody.
//UX prototype: Balsamiq.
//Simulation: MATLAB/Simulink.

Source code:
** Manage, access, build & test source code.
//IDE: Eclipse, PyCharm, IntelliJ.
//Static analysis: Pylint, Coverity.
//Build: Ant, Make, Maven.
//Unit Test: JUnit, pytest, NUnit, CppUnit.

Independent testing of software
** Manage & automate independent tests (see Quasar# 171):
//Manage: qTest, TestRail, Zephyr.
//Web: Selenium, Cypress.
//Desktop: WinAppDriver.
//API: SOAPUI, JMeter, Postman.
//Performance: JMeter, Gatling.
//Security: OWASP ZAP, Nikto, Linux security 'distros'.
//Test Environment: Docker, WireMock.
//Defect Management: BugZilla, MantisDB, Jira, Asana.
//Bespoke: Bash scripts, Python scripts, TCL/Perl scripts.

Build & Release:
** Source code access, build & deploy.
//Configuration Management: Git, Subversion.
//Continuous Intagratiion/Deployment: Jenkins, GitLab, Azure DevOps.
//Release Orchestration (complex environments): Harness, XL Release.

Customer Support tools:
** Help desk ticketing systems.
//ZenDeskm ServiceNow, DataDog.

Collaboration & documentation tools:
** Tools to host information sharing & history.
// Confluence, SharePoint, Doxygen, JDoc.
    
```

TABLE 2. COMMON AUTOMATION STRATEGIES

STRATEGY	STRENGTH	WEAKNESS
<p><b>Record and Playback</b> testing involves capturing user interactions with an application’s GUI and then automatically replaying these recorded actions during subsequent test executions.</p> <p>The process typically involves:</p> <ol style="list-style-type: none"> <li><b>1. Recording Phase:</b> The testing tool captures user actions (clicks, keystrokes, navigation).</li> <li><b>2. Script Generation:</b> The tool automatically generates test scripts based on recorded actions.</li> <li><b>3. Playback Phase:</b> The tool executes the recorded actions to verify expected outcomes.</li> </ol>	<p>Easy and fast to create scripts.</p> <p>Record and playback eliminate this barrier by removing the need for programming skills.</p> <p>Rapid Test Creation without extensive development time.</p> <p>Useful for getting an understanding of the programming logic.</p> <p>Useful for identifying GUI components.</p>	<p>Captures hard coded values (poor programming approach).</p> <p>Automation testers need to learn the supported language for automation.</p> <p>Very fragile as these scripts are sensitive to small changes in the GUI.</p> <p>Any change requires a re-recording for the script scenario across multiple scripts.</p>
<p><b>Structured Scripting</b></p> <p>Breaks down test functions into smaller, independent modules that can be reused across different test scenarios.</p>	<p>Uses software paradigm of reusable components.</p>	<p>Building an application to test an application – needs management and control.</p> <p>Depends on software engineering skills.</p>
<p><b>Data Driven:</b> Test data is kept separate from the automated test cases. The test data ‘drive’ the execution of functionality.</p> <p>For an example, a single test script can validate login and confirm username/ password combinations for error handling, based on the input data.</p>	<p>Robust.</p> <p>Increases reusability of script components in other tests by adding new input data.</p> <p>Reduced maintenance costs.</p>	<p>As for structured scripting.</p> <p>Complex scenarios require a ‘chain’ of scripts.</p>
<p><b>Keyword Driven:</b> Uses high-level keywords to represent test actions, creating a layer of abstraction between test cases and automation code.</p> <p>The intermediate layer of this approach translates what the keyword, e.g. ‘AppClose NoSave’, means in this specific case.</p>	<p>Test cases are readable and can be understood by non-technical stakeholders – allows ‘manual’ testers to ‘write’ automation tests.</p> <p>Ease of maintenance.</p>	<p>Increased set up efforts.</p> <p>Additional abstraction layers may cause performance issues.</p> <p>Requires significant design.</p>
<p><b>Hybrid Approach:</b> combines elements of data and keyword driven techniques (see case study).</p> <p>Consists of keyword layer, data layer and automated test layer.</p>	<p>Flexibility: combine explainability of keyword approach with flexibility of data approach.</p> <p>Facilitates non-technical team members.</p> <p>Ease of maintenance from modular and abstracted design.</p>	<p>Time required to design and build the design.</p> <p>Strong reliance on logging and self diagnosis to ensure maintainability.</p> <p>Requires good keyword design.</p>

## AUTOMATION TESTING STRATEGY (BEST PRACTICES)

1. Ensure automation is managed and controlled, with verification of its intended use. Leverage a SLC approach to the automation stack (scripts, tools, configuration management, defect management, environment management, reporting and so forth).
2. Define clear goals (what to automate, why)<sup>2</sup>.
3. Select suitable tools for system under test's technology stack (Selenium, Appium, JUnit, Robot Framework, etc.)<sup>4</sup>.
4. Ensure the correct skillset, the person who builds and maintains automation testing artifacts may or may not be a tester. The tester is usually someone with test skills and business knowledge but limited technical knowledge. The test automator should have technical software engineering skills. Here, the test automator supports the tester in the construction and maintenance of the automation<sup>4</sup>.
5. Framework Design: Data-driven, keyword-driven or hybrid depending on need<sup>9</sup>. Apply software engineering paradigms – write once, use in multiple scenarios: use parameterisation so that different data drives automation test script to run different test scenarios. Design the architecture of the automation tests to survive changes in the software GUI, such as the screen components or the language is changed to Japanese.
6. Prioritisation: Start with stable, high-value regression cases<sup>10</sup>.
7. CI/CD Integration: Automate in DevOps pipeline for continuous validation<sup>8</sup>.
8. Self-diagnosis: Ensure detailed logging/reporting<sup>4</sup>.
9. Maintainability: Review, update and retire outdated scripts regularly<sup>6</sup>. Ensure that there is ongoing maintenance to avoid fragile or abandoned test suites<sup>10</sup>.
10. Human-in-the-loop: Always combine automation with manual testing<sup>6</sup>. As with AI – provide explainability. For example, demonstrate how the testing is sufficient, what was tested, test case change management and reporting.

## CONCLUSION

Testing involves the selection and execution of a subset of a vast number of possible test scenarios within a limited timeframe. This subset of tests is expected to find the greatest number of defects in the system under test while also providing confidence that it will work.

Automation can reduce the effort required to execute tests and it can significantly increase the volume of testing within a set timeframe. It is a skill that is different from manual testing.

[Auditor check] Ask about the education, experience and training of the automated testers in skills beyond the QMS and GxP training.

Automation tests are static, they permit the re-execution of the exact same test steps, with the exact same input data time and again. Something that is not guaranteed with manual testing. When managed diligently, it can be a mechanism to reduce the scope of human execution error and increase the scope of checks on the level of product software quality.

Manual and automated testing complement each other. Automation is not a replacement – it frees testers from repetitive work so they can focus on more complex, value-added testing<sup>9</sup>.

Automation costs more to implement than manual tests, but the cost of execution (once deployed) is a fraction of a manual execution. Hence it is a goal of suppliers and regulated users.

Some organisations will attain significant amounts of savings, for example by leveraging machines out of hours when they are idle. Organisations achieve better software product quality more quickly. Other organisations may have minimal improvements. This is dictated by the testing strategy and the maturity of the associated QMS. For example, using automation to run the same narrow tests consistently will not provide value add or confidence.

The authors have briefly touched on this software engineering discipline. There is more to the automation testing than just GUI level UAT testing. Similarly, there is more to automation than just a testing tool. The CS auditor needs to gain an understanding of the strategy and discern between the 'marketing' and the facts.

## REFERENCES

1. Computer Software Assurance for Production and Quality System Software, FDA,2022
2. ISTQB Foundation Syllabus, Software Testing Foundation, ISTQB, 2018.
3. Standard for Software Test Documentation, IEEE Standard 829-2008.
4. Software Test Automation, Fewster & Graham, 1999
5. General Principles of Software Validation, Final Guidance for Industry, FDA 2002.
6. Exploratory Testing, Kaner IEEE Computer, 2006.
7. SeleniumHQ Documentation – [www.selenium.dev/documentation](http://www.selenium.dev/documentation)
8. Applttools. Automation Testing Guide – <https://applttools.com/>
9. Guru99, Automation Testing – What is, Benefits, and Types – [www.guru99.com/automation-testing.html](http://www.guru99.com/automation-testing.html)
10. The Art of Software Testing, Myers, 2011.
11. "Research on software testing techniques and software automation testing tools." IEEE Conference Publication, doi: 10.1109/ICRISET.2018.8389562, IEEE (2018).
12. The Art of Software Testing, Myers, 1979.

## 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.

## CASE STUDY: CTMS VENDOR

### EXAMPLE TEST CODE

CTMS company utilises the Cucumber automation tool to implement GUI level automation. Cucumber was selected as utilises ‘Gherkin’ notation to write test scenarios in natural language that is easily understood by technical and non-technical personnel.

- **Feature:** (describes the feature under test in a user story format)
- **Background:** pre-test steps that apply for all the test scenarios in this feature
- **Scenario:** individual test case
- **Scenario outline:** template for multiple scenarios outline. Enabler for data driven testing
- **Test step keywords** to execute the test: **Given** (context), **When** (action being performed), **Then** (expected outcome), **And/But** (continues previous step).

The following test code provides an example, showing:

**Keyword driven** – apart from ‘Scenario Outline’, each phrase has underlying code that achieves the specified state or executes the specified action. The phrases can be re-used in different scenarios, so that ‘the user is signed in’ will just run code that signs a user in.

**Data driven** – the test scenario is run for each of the data values for the variable ‘issue\_status’ in the list under the ‘Examples:’ header

```
Scenario Outline: Cannot close an issue if there are one or more opened action items
Given there are multiple opened action items on that country issue
And the user has been granted the role "custom role" on study
And "custom role" can perform activity "manage_issues" with permission "read update"
And "custom role" can perform activity "manage_action_items" with permission "read update"
And the user is signed in
When they access the edit page of the specific country issue directly
And they choose the new issue status as <issue_status>
Then a warning pop up will appear on the screen confirming the issue cannot be closed as there are opened action items on it
And the user returns to the previous edit page that has the initial opened issue status
Examples:
| issue_status |
| Closed - Cancelled |
| Closed - Duplicate |
| Closed - Resolved |
```

### Example of an underlying step definition:

```
When('they navigate to the issue edit page') do
  if @issue.level == 'study'
    visit studies_issues_pages_path(@study, page: @issue.id.to_s)
  elsif @issue.level == 'country'
    visit edit_manage_issues_study_country_issue_path(@study, @study_country, @issue)
  elsif @issue.level == 'site'
    visit edit_manage_issues_study_site_issue_path(@study, @site, @issue)
  else
    raise "Unknown issue level: #{@issue.level}"
  end
end
```

This example demonstrates the transition from hard coded values, e.g., ‘manage issues’ into parameterised values e.g. ‘<issue status>’.

## TEST MANAGEMENT SYSTEM EXAMPLE

The following screen extract of an automated test run report demonstrates the immediate feedback loop from a cloud test platform linked to the source code system. The cloud test platform runs all the automated tests on every new build of the source code.

Each test run reported with outcome:

- Green highlighted is the unique ID of the version of the codebase tested
- Red highlights the failed tests – the user can drill down to get detailed information including screenshots.

The development team’s rule of ‘never break the build’ ensures that immediate action is implemented to resolve the software product quality issue.

In this case 32 failed tests related to a feature permission. This example demonstrates the use of automation in a build to provide feedback on the quality of code updates submitted on the most recent code change. The ‘self policing’ nature of the group enforces that the developer in question will seek to troubleshoot and resolve the issue before it beds in and dilutes the software product quality.

