How to Fix 'Test Automation' for Faster Software Development at Higher Quality

Matt Griscom

matt@metaautomation.net

MetaAutomation LLC

Abstract

From years in the trenches of "test automation," I found it broken. Applying "automation" to "test" creates mismatches leading to inefficiency, poor communication, and quality risk. I discovered and recorded the MetaAutomation pattern language to fix it and make quality feedback much faster, more reliable, richer, and more valuable across the larger team and distributed teams. The base pattern is simple and familiar: it enables self-documenting checks with complete details driving and measuring the System Under Test (SUT), creating very trustworthy structured artifacts that give deep knowledge of SUT behavior for rich communication and analysis. Any team member can drill down through the hierarchy from business-facing steps to technology-facing steps of the check result. Every completed or failed step shows milliseconds-to-completion, and the artifact shows blocked steps too. The hierarchy supports data-driven timeouts for any step. Manual testing becomes more fun and effective with complete clarity on automation actions and results. MetaAutomation shows how to maximize scale. It solves the flaky check problem, cuts defect escapes, and on check failures it sends directed notifications to team member(s) that need to know so issues needing attention from a team member are resolved at correct priority. This paper discusses how to prioritize and optimize automated checks and how to fill out the quality automation problem space to empower the QA team as the glue that binds the larger team together. Working samples on metaautomation.net show single-process, multi-process, and multi-tier checks.

Biography

Matt Griscom has 30 years' experience creating software including innovative test automation, harnesses, and frameworks. Two degrees in physics primed him to seek the big picture in any setting. This comprehensive vision and a love of solving problems that are important yet challenging led him to create the MetaAutomation pattern language to address the quality automation problem space of the software business. Matt blogs on MetaAutomation, consults, and recently published his third book on MetaAutomation. He also hikes, kayaks, runs, helps run an awesome Toastmaster group, and lives with his wife and two kids in Seattle. Matt loves helping people be more effective in doing software quality and is available by email at matt@metaautomation.net.

1 Introduction

Consider these four questions:

1. Does the SUT do what we need it to do, and do it fast enough?
2. Can automation verify requirements of behavior and perf on the SUT, and do it fast and thoroughly enough to ensure that quality always moves forward?
3. Can the whole team access highly detailed and trustworthy information on that, in role-appropriate ways?
4. Can we notify the developers and QA people who need to know, while avoiding non-actionable emails and improving response times to un-block the team as needed?

This paper, and the corresponding talk, is about enabling “Yes!” to the questions. The journey requires questioning some common understandings and practices.

1.1 A Better Way

Imagine the state of “test automation” applied currently to measure and report quality on the SUT.

By contrast, imagine an ideal solution for the business with automation: measuring, recording, and reporting detailed information to fulfill those four questions. This ideal solution reconsiders everything that we know or think we know about the profession and practice.

It helps to start with the “big picture” and define the space for the problem at hand. I call this problem space “quality automation,” with automation to serve quality between the technology-facing interface of driving and measuring the SUT, and the business-facing interface of serving information to the people of the business. Quality automation serves the interests of software quality: driving it forward, while managing quality risk, with automation where that applies to answer the four questions.

1.2 Break with the Past

Zooming in from the big picture to the component details shows that some practices are broken.

First, there was Glenford Myers’ mistake with his 1979 book “The Art of Software Testing.” Myers was explicit:

Testing is the process of executing a program with the intent of finding errors. (Myers, 1979, p. 5)

He elaborates on the topic at length, including this on the next page:

… since a test case that does not find an error is largely a waste of time and money… (Myers, 1979, p. 6)

I summarize the mistake as “Testing is only about finding bugs.” The book was published so long ago, and the impact of software on people’s lives was so drastically different then as compared to now, that the mistake was a very minor one at that time and approximately correct – yet flawed. The importance of the flaw was magnified in the following 40 years, while the influence of such an important book from so long ago was magnified as well.

2 The Problem with “Test Automation”

Applying SUT automation to test birthed the obvious phrase to describe that union: “test automation.” Unfortunately, the implication of the phrase — that automation produces the same value as manual testing, just faster — lives on today. Automation applied to functional quality for “test” does not work like industrial automation does for building furniture, or flight deck automation does to help pilots fly a plane; it is very different.

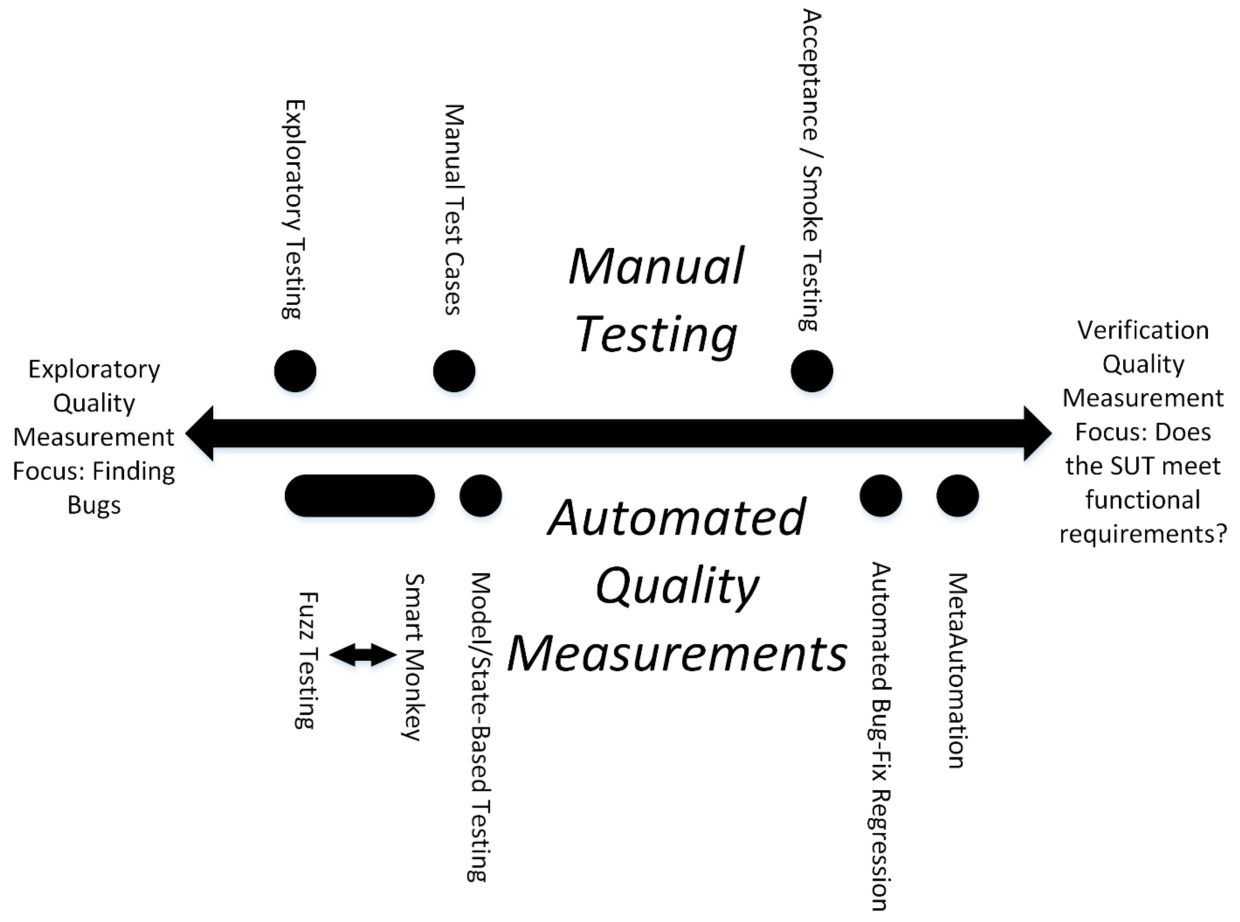

Exploratory Testing
Manual Test Cases
Manual Testing
Acceptance / Smoke Testing
Exploratory Quality Measurement Focus: Finding Bugs
Verification Quality Measurement Focus: Does the SUT meet functional requirements?
Automated Quality Measurements
Fuzz Testing
Smart Monkey
Model/State-Based Testing
Automated Bug-Fix Regression
MetaAutomation

Figure 1. Two important focuses for automation and manual testing, and how some practices align towards one or the other

By linguistic relativity (a.k.a. the Sapir-Whorf hypothesis) “test automation” locks us into a very expensive misunderstanding about what automation can do for quality. This misunderstanding limits the productivity of the QA role and minimizes their importance to the larger software team.

The need for manual test will never go away because humans are unmatched in perceptive intelligence towards deciding whether some bit of SUT behavior is actionable or not (and, the rise of AI will not replace people for at least a decade). But, automation can be very valuable, and much more so if it is not constrained by misunderstandings.

Therefore, I use the phrase “quality automation” to describe the problem space (as discussed above) *and* the service of automation towards bringing value in that problem space. Quality automation is much broader in capabilities than “test automation” ever was or ever could be and does not have the baggage. One could replace the mistaken phrase “test automation” with “SUT automation” but since automation around reporting and communication depends so closely on what “SUT automation” does, I prefer to just replace “test automation” with “quality automation.”

As a side note, there are other positive movements in software quality that are complementary to quality automation: Analytics and A/B testing remain powerful and important trends. Shift-left testing is a clever idea to getting quality results faster and closer to developers — if system testing is not neglected.

My personal favorite is bottom-up testing: system testing done first at the least dependent layers of the SUT, e.g., a database layer with no mocks, and then working up to the more dependent layers, e.g., a GUI. This is powerful for quality automation, because higher-impact and higher-risk issues are found much faster this way. Complete system testing can be addressed at a lower priority because issues found in the most dependent layers are low-risk to change or fix.

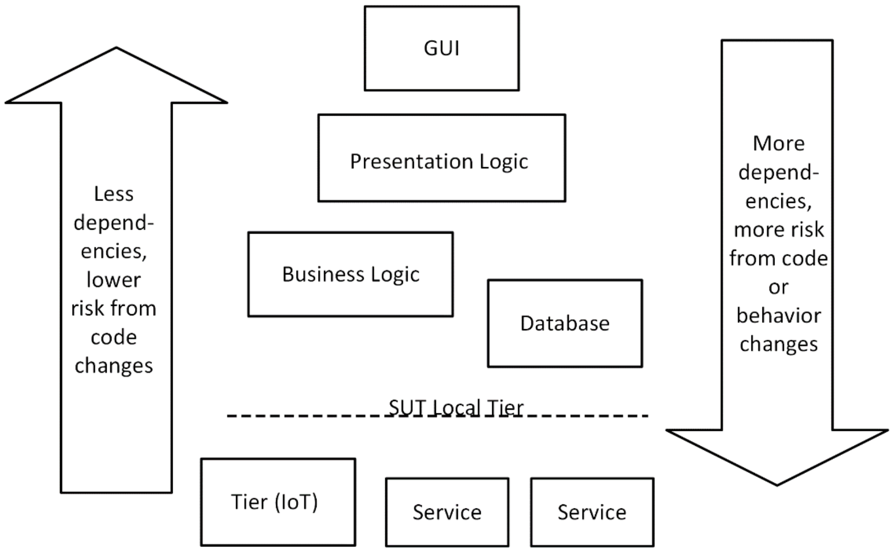

GUI
Presentation Logic
Business Logic
Database
SUT Local Tier
Tier (IoT)
Service
Service
Less depend-encies, lower risk from code changes
More depend-encies, more risk from code or behavior changes

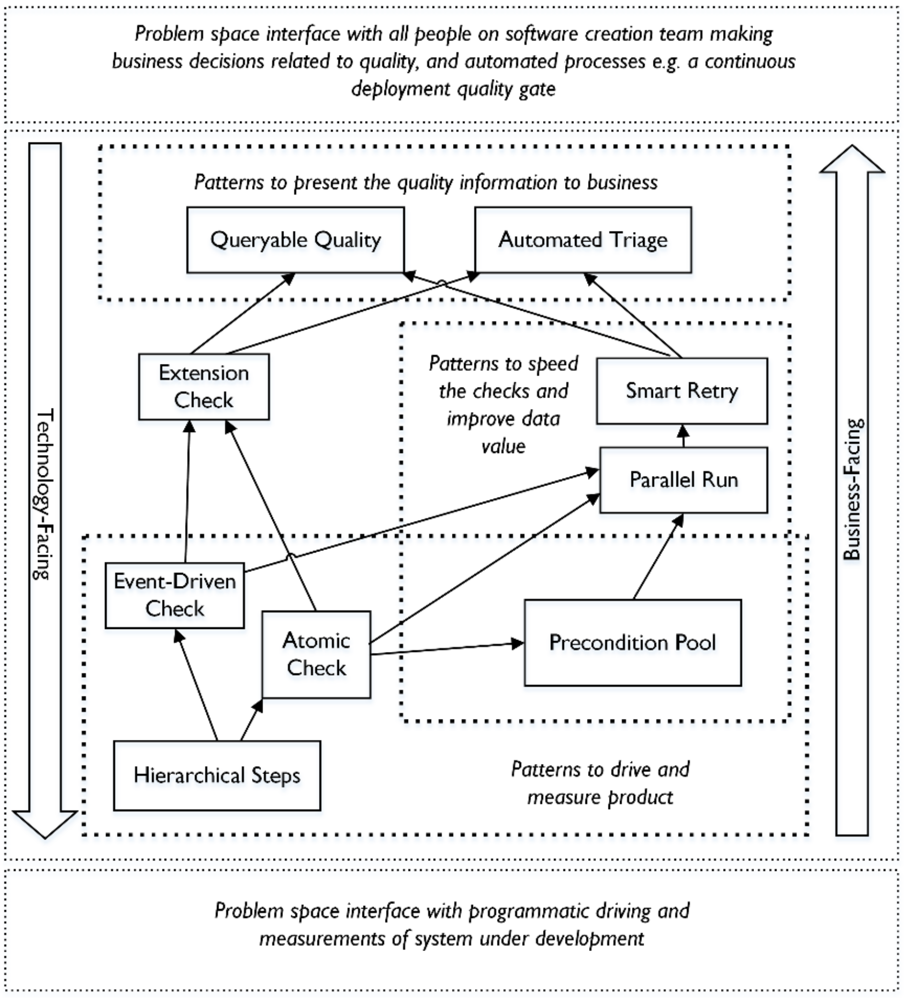
Figure 2, Dependency vs. quality risk

The focus of this paper is on repeatable checks, because those are of core importance to getting trustworthy quality results fast. The Hierarchical Steps application towards SUT automation, which creates a structured record of all data on driving and measuring the SUT, is also generally applicable towards model-based testing or other “big” tests with automation.

3 MetaAutomation

Simply put, patterns are solutions to a problem in a context. A prescriptive pattern language is a set of patterns with intra-pattern dependencies to address a complete problem space.

MetaAutomation is a pattern language that describes an ideal platform-independent, language-independent solution to the quality automation problem space. All patterns in MetaAutomation are based on existing practices, but it combines them in ways both old and new to achieve the ideal. It is not an unchanging pattern language; it will become a living pattern language with extensions and refinements as needed with input from the community it serves.



Problem space interface with all people on software creation team making business decisions related to quality, and automated processes e.g. a continuous deployment quality gate
Patterns to present the quality information to business
Queryable Quality
Automated Triage
Extension Check
Patterns to speed the checks and improve data value
Smart Retry
Parallel Run
Event-Driven Check
Atomic Check
Precondition Pool
Hierarchical Steps
Patterns to drive and measure product
Technology-Facing
Business-Facing
Problem space interface with programmatic driving and measurements of system under development

Figure 3, the MetaAutomation pattern map filling the quality automation problem space

Everybody uses the Hierarchical Steps pattern, whether they work in software or not; it is a natural way of describing, communicating, or executing any repeatable procedure. Applying this pattern to quality automation is the most fundamental — and radical! — change that MetaAutomation recommends. Implementing it is not trivial, so working open-source samples are available on GitHub, linked from the MetaAutomation.net web site.

Milliseconds-to-completion for every step in the hierarchy is automatically recorded. In case of check failure, the failed steps going from leaf step up to root are noted, and blocked steps show the quality risk of unmeasured SUT behavior. Each step can have a data-driven timeout for that step, too, configured directly in the artifact of a check run to serve as a guide for the next run of that check.

A check is a kind of test that is suited for automation; there are no extra steps or measurements, nor any assumption of powers of observation that a person might bring to a manual test but that SUT automation cannot do.

Atomic Check is familiar already to people who are skilled at SUT automation for quality. Simply put, it is about measuring a functional requirement in a way that is as simple as possible. Atomic checks have priority from functional requirements (from prioritized business requirements) first and implementation order second.

Event-Driven Check is like Atomic Check, but where the SUT is triggered by events that cannot necessarily be controlled in the same way as, e.g., API or service calls.

Precondition Pool is about managing check preconditions outside the check, to simplify and speed the check. This pattern is a generalization of the pattern of managing such check preconditions as environments in which checks are run, to also managing user accounts, documents, databases, or anything else the check needs but which is ancillary to the focus of the atomic check and can be managed asynchronously and out-of-process.

Parallel Run is widely used; it is just running checks in parallel across multiple processes and/or machines.

Smart Retry is like the Retry pattern where a check is retried on failure, but smart because the data (from applying the Hierarchical Steps pattern) is readily available to the automation to determine root cause of a failure, whether it as just been reproduced, and whether a retry is in order, e.g., in case of a timeout on a GUI thread or external dependency. When the Smart Retry implementation is correctly configured, it solves the flaky-check problem.

Extension Check is a check based on the detailed artifacts (results) from an earlier check, for example, to verify that certain performance criteria are met.

Automated Triage directs notifications based on root cause of an actionable failure. For example, it might notify a developer if it is determined – based on the artifact of a check, or a reproduced failure – that a developer “owns” the feature involved with that root cause.

Queryable Quality is an implementation of an intranet site to show all the data that quality automation provides on the SUT, including behavior, verifications, the performance measurements with every step, etc. This is like an information radiator for QA, but much richer in data and highly interactive. This is where team members might review a run of a set of checks or view the steps of a check or checks and drill down from the business-facing steps near the root to the technology-facing leaf steps that drive and measure the SUT.

4 Conclusion

Coming back to those four questions:

1. Does the SUT do what we need it to do, and do it fast enough?

For software that matters to people, it is both unethical and poor business practice for the team to ship anything that does not meet these criteria (other than explicitly alpha or beta software). Quality in software is becoming more important every year. The mistakes and inefficiencies of “test automation” are holding software quality back; software teams need a smarter approach to ground their quality efforts.

It is time to leave behind any illusion that manual test and automated verifications are effective at the same things.

2. Can automation verify requirements of behavior and perf on the SUT, and do it fast and thoroughly enough to ensure that quality always moves forward?

By using the wrong tool for the job — log statements — conventional “test automation” drops valuable information of SUT behavior on the floor, so it cannot answer the question in a detailed or even trustworthy way. Even BDD automation drops any information behind the keywords. Starting with the Hierarchical Steps and Atomic Check patterns, MetaAutomation shows how to record and present this information with no need to even look at the code, cut false positives, cut defect escapes and non-actionable SPAM notifications, and do it with specific guidelines and techniques to do it as fast as possible.

3. Can the whole team access highly detailed and trustworthy information on that, in role-appropriate ways?

The structure and detail of Hierarchical Steps extends from the business-facing steps at and near the root of the hierarchy, to the technology-facing steps at the leaf nodes. On an intranet site, team members can drill down into details of SUT behavior and verifications as needed to get an unprecedented view into behavior and performance.

For manual testers, it means that they do much less repetitive testing because they know exactly what the quality automation system did with the SUT, and they know exactly where exploration is needed. Manual testing becomes more fun and more effective.

For the whole team, defect escapes are reduced because the details of what is measured is highly trustworthy and accessible; what is not measured is discoverable by comparing SUT behavior with the quality portal that implements the Queryable Quality pattern.

4. Can we notify the developers and QA people who need to know, while avoiding non-actionable emails and improving response times to un-block the team as needed?

MetaAutomation shows how to record and communicate the detailed, structured, and highly trustworthy information, and the solutions that are needed to make these things happen. It shows the QA role how to assume the central role that they deserve: the backbone of communication and collaboration around SUT behavior, and the role that enables the larger team to ship higher-quality software faster.

References

Myers, Glenford. 1979. *The Art of Software Testing*. John Wiley & Sons, Inc.