MetaAutomation: A Pattern Language to Apply Automation to Software Quality

MATT GRISCOM, Principal, MetaAutomation LLC

MetaAutomation is a pattern language for automated measurements and communication of functional software quality and performance within a team or company that is developing software. The "whole" of MetaAutomation addresses the quality automation problem space: automation to measure and communicate quality, bound on the technology side by operations on and measurements of software under development or maintenance for quality purposes, and bound on the business side by human customers of the quality information, and other automated processes that depend on quality, for example, operations.

The focus of MetaAutomation is on answering the question for the business "Does the system do what we need it to do?" quickly and reliably, with highly trustworthy and structured detail that supports unprecedented visibility into and communication around the larger team – including dev, QA, and every team member concerned with quality - of what the automation drives the product to do, and how the product responds.

Each of the patterns is based at least in part on existing patterns of human behavior and/or software development.

MetaAutomation clarifies the value of what an intentional, designed approach to measuring and reporting software quality with automation can achieve, as opposed to common patterns of doing this which make poor use of automation's capabilities.

The target audience is more than just the quality assurance role; it is anybody doing, managing, or leading quality work with automated verifications and communication of functional and performance requirements on a software development project, including software developers who would like to create software faster and with higher confidence.

MetaAutomation has 9 patterns currently: Hierarchical Steps, Atomic Check, Event-Driven Check, Extension Check, Precondition Pool, Parallel Run, Smart Retry, Automated Triage, and Queryable Quality. It is open to extension with more patterns that address the quality automation problem space, and a community to make it the living pattern language that it deserves to be.

Categories and Terms from ACM CCS 2012: **[Software and its engineering]**: Software creation and management—Software verification and validation—Software defect analysis—*Software testing and debugging*; **[Software and its engineering]**: Software creation and management—Software verification and validation—Process validation—*Acceptance testing*; **[Software and its engineering]**: Software creation and management—Software development process management—*Risk management*

General Terms: Software Quality, Quality Automation, Automation, MetaAutomation, Test

ACM REFERENCE FORMAT:

Griscom, M. 2017. MetaAutomation: A Pattern Language to Apply Automation to Software Quality. HILLSIDE Proc. of Conf. on Pattern Lang. of Prog. 24 (October 2017), 34 pages.

1. INTRODUCTION

Software quality is a very open-ended pursuit, because there are many different views and values on what quality means for the pure-information domain of software. Perfection in quality is elusive. Depending on the problem or issues addressed, the importance of quality ranges from significant, e.g., for a game on a mobile device, to pivotal for the business, e.g., in the domain of planes, rockets, or self-driving cars. As software and information become ever more important to people's lives, software quality also becomes ever more important.

The pattern language MetaAutomation concerns the functional and performance software quality domains, including such aspects as reliability and trustworthiness, from the positive perspective summarized by the question "Does the system do what we need it to do?"

Sample implementations available on http://metaautomation.net show the Hierarchical Steps, Atomic Check, and Parallel Run patterns, expressed in running code. For further information and clarification on some concepts and benefits covered briefly in this paper, please reference these samples, and consider building, running, changing, and reusing them. A free version of the current Microsoft Visual Studio is enough, but most of the sample code, and the entire MetaAutomation concept, is platform-independent.

1.1 Why a Pattern Language?

The 9 patterns of MetaAutomation are more than a list or a catalog; they form a structure with defined dependencies, and together they define a whole, a coherent solution to a problem space I call "quality automation:" how best to drive automated measurements of functional and performance software quality and communicate the quality to stakeholders of the business, both human and automated processes, fast and often. With quality automation, the quality measurements, recording, re-measuring, directing and presenting communication are all potentially automated, working from the least dependent patterns up as they make sense for the software developing organization. The more dependent patterns form a strong, clear expression of business value to motivate implementing the less dependent patterns [Leszak et al. 2000].

Common misunderstandings and practices around software quality are preventing software from achieving levels of quality that are necessary today, and crucial tomorrow. Current, pervasive, and very costly misunderstandings include the persistent but obsolete meme that "The point of test is to find bugs" [Myers 1979]. There is also commonly a poor understanding, or even a complete discounting, of the fact that automated verifications can excel in a very different business value than manual testing offers.

Traditional yet limiting practices include using the Linear Logging antipattern. This appears as linearly occurring log statements with inherently weak contextual information, in procedures of automated verifications where context is actually very important; logs work great for isolated events (e.g., with web servers), but they are very poor for conveying context of a step in a procedure.

MetaAutomation solves the problems of misguided design and lost information for automated quality measurement and communication.

As a pattern language, MetaAutomation clarifies the nature and boundaries of the quality automation problem space for the software business and clarifies how teams might implement solutions. It highlights the benefits of taking an enlightened, creative approach to quality automation rather than the historical limitations of current practices.

1.2 Why "MetaAutomation?"

The "Meta" of MetaAutomation invites a broadening of perspective, a more abstract, general, and high-level view of applying automation to software quality. It begins by asking the question: if we do apply automation to software quality, what can we learn from first principles and the big-picture view to deliver the greatest quality-management value to ship software faster?

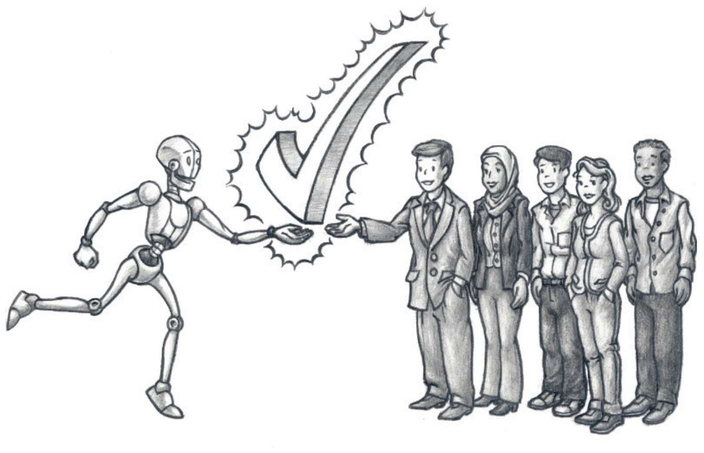


Fig. 1. MetaAutomation does not necessarily change how developers write product code, but it does show the QA role how to enable faster development with higher quality confidence.

Automated test cases or scenarios are more effective at assuring quality than finding bugs, but both are important for software development. MetaAutomation takes quality assurance to the next level by laying out the value of what is possible. Part of that clarity comes from differentiating between best techniques to answer the questions "Does the system do what we need it to do?" vs. "Are there bugs for us to find?"

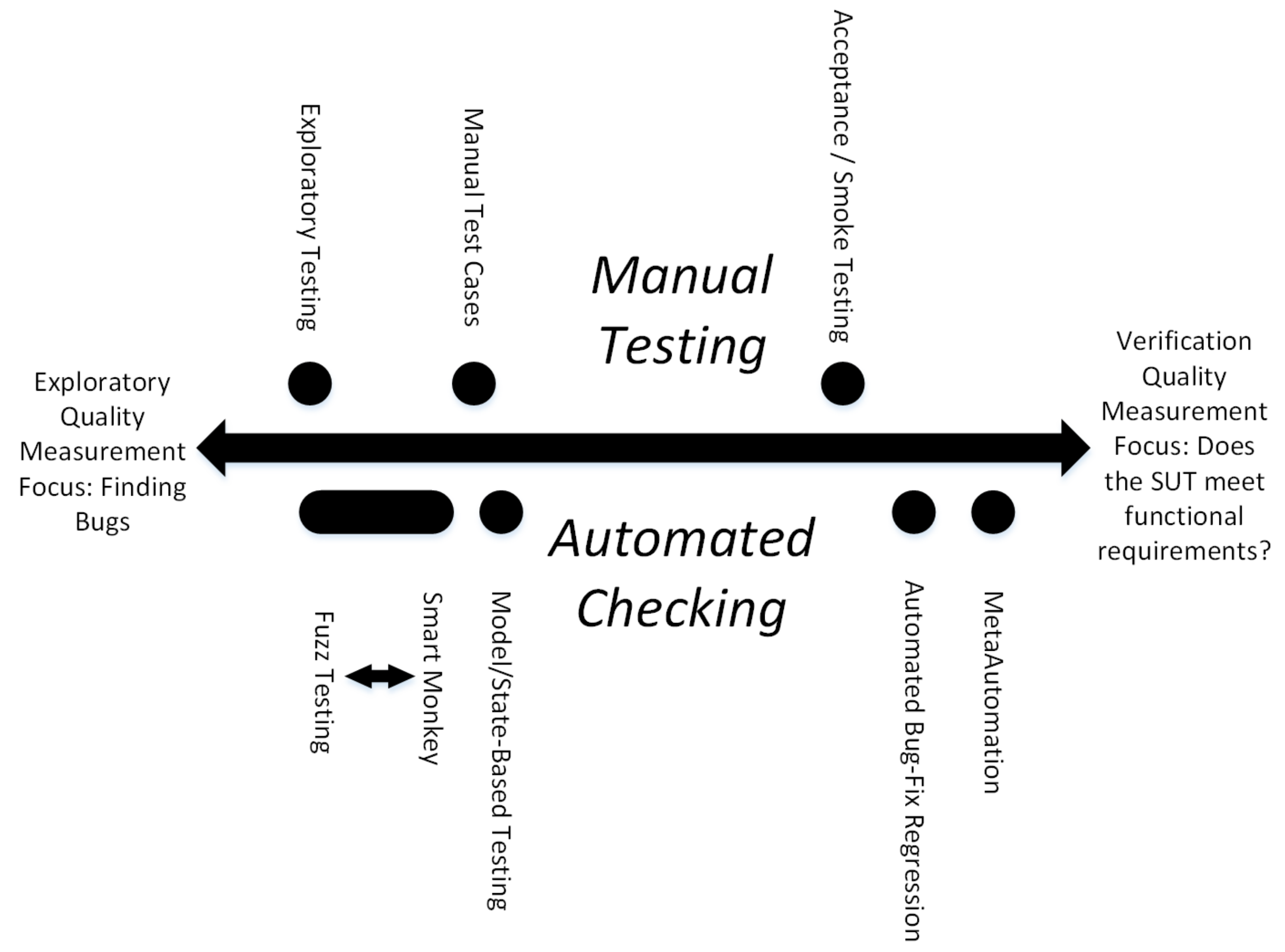

Exploratory Testing
Manual Test Cases
Manual Testing
Acceptance / Smoke Testing
Exploratory Quality Measurement Focus: Finding Bugs
Verification Quality Measurement Focus: Does the SUT meet functional requirements?
Automated Checking
Fuzz Testing
Smart Monkey
Model/State-Based Testing
Automated Bug-Fix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

Fig. 2. MetaAutomation compared to other types of functional testing.

1.3 Overview

The pattern map (Figure 3) shows the 9 patterns of MetaAutomation, their names and grouping to clarify the function of each of the patterns in the larger pattern language context, and the problem space bounds – the interfaces with the business and the software under development.

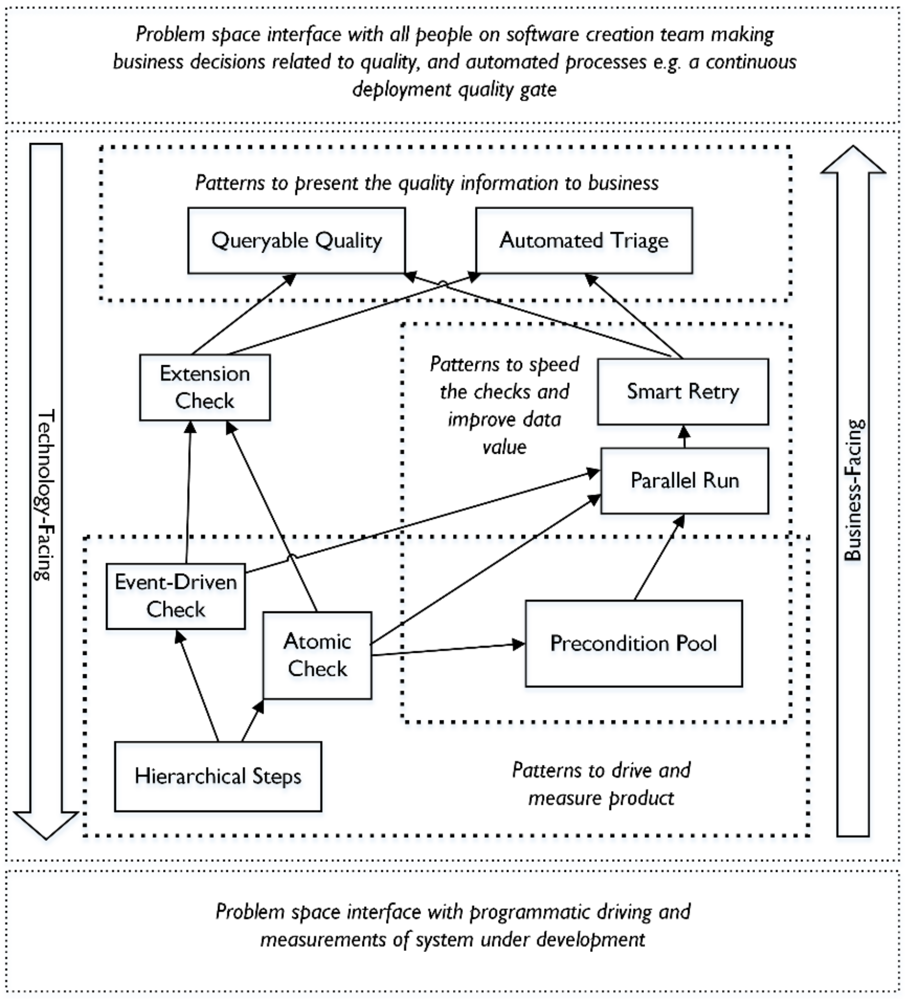

Problem space interface with all people on software creation team making business decisions related to quality, and automated processes e.g. a continuous deployment quality gate
Patterns to present the quality information to business
Queryable Quality
Automated Triage
Extension Check
Patterns to speed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

Fig. 3. MetaAutomation Pattern Language Map.

At the bottom of Figure 3 is the technology-facing context; this is how MetaAutomation relates to the code of the system under development, technological dependencies that the product has, and/or technologies used to drive the system for quality purposes.

At the top of Figure 3 is the business-facing context: people on the team making business decisions related to quality, and any automated processes outside of quality that drive the business, e.g., promoting a system build as part of software continuous deployment.

The following table summarizes the patterns of MetaAutomation as presented in this paper.

Table 1 Summary of MetaAutomation Patterns

Pattern Name	Problem	Solution
Hierarchical Steps	How to record all interaction details with the SUT to support query and analysis?	SUT-driving code to create at runtime an extensible hierarchy, like XML.
Atomic Check	How to maximize the value, trustworthiness, speed, and scalability of quality data?	Make each check independent, and drive the SUT to verify one requirement or cluster per check.
Event-Driven Check	Events drive the SUT, internally and externally. How to get fast, repeatable,	The SUT subscribes to external events from the automation. Verifications wait on events as needed.

	trustworthy measurements of functional quality?	
Extension Check	Fast and trustworthy feedback is very important, even on quality issues that cannot be driven deterministically.	Instrument or measure the SUT on the non-deterministic quality aspect, and store the data in the artifact of another check. Run verification(s) on the data created by the first check.
Precondition Pool	Is there a way to boost check performance, make checks more reliable, and improve check run data?	Actively manage preconditions out-of-line in a pool or pools.
Parallel Run	The business needs to run many checks quickly.	Distribute runs of the individual checks across (virtual) machines or environments.
Smart Retry	False positives, and check failures that the team must reproduce, are costs to the business.	For configured root causes of a fail, immediately retry the check for a total of 2-3 tries, and group resulting data.
Automated Triage	Notifications of check failures sent to large sets of people are a cost to the business and decrease trust in the quality automation.	Send actionable notifications only to those who would act.
Queryable Quality	MetaAutomation creates detailed and trustworthy data on SUT behavior and performance. This must be available and useful to anybody on the team.	Create a link-able, query-able interactive portal to show the data, starting with business-facing details and allowing drill-down to technology-facing details.

Begin design and implementation following the Hierarchical Steps and Atomic Check patterns.

1.4 Terms

Following are some terms clarified for the context of this paper.

1.4.1 Actionable

A work item for a person on the software team that has clear business value for the software product and clear next steps is actionable. For example, a bug assigned to a developer with clear root cause and steps to reproduce the problem is actionable, whether the bug is ultimately fixed or not.

1.4.2 Antipattern

An antipattern is a common pattern of behavior in response to a recurring problem that has a significant negative attribute. For example, the Chained Tests pattern recorded by Meszaros is an antipattern in the sense that chaining automated tests has significant negative impacts on efficiency, value of data generated, and the scalability of the check runs with computing resources [Meszaros 2007, p. 454].

1.4.3 Artifact

An artifact is information generated as part of the software development process that is not part of the software product. For the context of this paper, an artifact is information on quality of the SUT that is measured and recorded while executing a bounded and repeatable automated check.

1.4.4 Atomic Check

An Atomic Check verifies the linked functional requirement using as few steps as possible in driving the SUT. To avoid the risk of measuring the SUT out of context, all dependencies are in place.

See Check.

1.4.5 Atomic Step

An Atomic Step is a step in a procedure that, from the perspective of non-product code or code owned by the QA role, cannot divide into smaller non-trivial steps (i.e., steps that could fail). In the hierarchy of steps for an Atomic Check, the atomic steps are also leaf steps because, being indivisible, they have no child steps.

1.4.6 Automated Checking

With automated checking, fail events can only come from the SUT, or quality code.

Contrast with Manual Testing.

1.4.7 Business

For the context of this paper, "business" or "the business" refers to the higher-level purpose of the software project or system or organized team of people. This could be, e.g., a for-profit business, or an embedded software system, or an open-source software project. "The business" could therefore be the equivalent of "the high-level method that the team uses to deliver the greatest value."

1.4.8 Business Requirement

A business requirement is a required characteristic of the SUT, defined in an implementation-independent way and from a customer or business perspective. Business requirements link to functional requirements.

See Atomic Check.

See Functional Requirement.

1.4.9 Check

Check is a type of test where verifications are explicitly coded or implicitly verified as part of the code that executes the test. This excludes, for example, the exploratory part of any tests driven by people, because people see issues with the SUT that might not have been targeted in advance.

See Manual Testing.

1.4.10 Cluster

See Verification Cluster.

1.4.11 False Negative

If a check passes, there is no alert; it is a negative event. If at some future point it becomes clear that the check should have failed, the negative is therefore false.

1.4.12 False Positive

A check failure that is not actionable is a false positive: a positive alert signal (fail), but false (not actionable).

1.4.13 Functional Requirement

A functional requirement is some required, measurable behavior of the SUT. Functional requirements connect to Business Requirements.

See Business Requirement.

1.4.14 GUI

Graphical User Interface.

1.4.15 Manual Testing

With manual, or human-guided testing, it is a person that decides whether some aspect of or issue with the SUT should be recorded and/or promoted as an action item or bug (whether the person is using any tools).

Contrast with Automated Checking.

1.4.16 Quality Automation

Quality automation is automation to support functional and performance quality as part of the software development process. The scope of quality automation includes:

- driving the SUT for quality measurements
- making those measurements
- recording the procedure and measurements
- improving business value of the quality data
- making both directed (i.e., push) and queryable (i.e., pull) communications of that data to the software business

Note that quality automation does not necessarily influence the style, contents, design, or language of product code.

The customers of quality automation include both people doing the software business, including developers, and automated processes, e.g., for continuous deployment of software.

Quality automation is the best method for measuring and communicating quality to the business, to answer these two questions:

1. Does the system do what we need it to do, for functional and performance quality measures?
2. By those measures of the first question, is quality for the SUT always getting better (or at least measurably the same), down to the control and granularity of individual code change submissions?

Quality automation code has no direct impact on end-users of the SUT, so coding styles and standards can have significant differences from product code.

See also Automated Checking.

1.4.17 System under Test (SUT), or System

The system under test (or, system for short) includes product code owned by the team developing the software. It excludes dependencies that are outside team ownership, and non-shipping code. Product code is code that will touch or impact end-users and any external dependent software systems.

1.4.18 Verification

In software quality, verification is a measurement of whether the SUT meets a functional requirement. This is like a "test" but specifically focused on predetermined success criteria that automation can measure.

1.4.19 Verification Cluster

A group or cluster of more than one verification. This is a useful optimization for cases where more than one related functional requirement can be verified with no intervening interaction with executable code of the SUT. For example, on a single static web page, verifications of several required properties of similar priority would fit in a verification cluster. The result of a verification cluster is a Boolean pass/fail, with information on each verification in the cluster. A failure in a verification cluster never blocks any of the measurements in the cluster.

2. THE PATTERNS OF METAAUTOMATION

Following are definitions for the 9 patterns of MetaAutomation. There will be more patterns in future (see section 2.10) as people recognize the need to address SUT's of different architectures, and diverse needs, within the quality automation problem space.

2.1 Hierarchical Steps

2.1.1 Summary

This pattern is about using an ordered-tree hierarchy to record and communicate a repeatable procedure.

Fig. 4. Four levels of hierarchy represent the robot's four-day journey: the robot's steps, the boulders, the camping tents, and the house as ultimate destination.

2.1.2 Problem

When driving the SUT for quality measurements, the details of interacting with the SUT are as important as the ultimate measurement results [Davies et al. 2014]. This is true whether a check passes or fails. Both types of information are valuable to the business [Parveen et al. 2007]. Together, if persisted and handled efficiently, they can help the rest of the quality automation process reach much greater speed and efficiency, and become more effective at ensuring that quality always moves forward.

More precisely, knowledge of how automation drives the SUT can prevent

- False positives
- false negatives
- low trust
- poor communication about SUT behavior, especially across teams, roles, and geographies
- re-work by people doing manual testing in coordination with the automation.

Without those details, for the team to ensure that quality is always moving forward is difficult and expensive, if it is possible at all [Herzig and Nagappan 2015] [Jiang et al. 2017] [Elbaum et al. 2014].

Unfortunately, conventional practices that rely on scattered log statements in non-shipping automation code are very lossy and therefore poorly suited to recording how automation drives and measures the SUT, especially if more quality automation consumes the artifacts from the check runs. Such practices lose most of the business value of the data.

2.1.3 Context

This pattern applies whenever the "how" of a task is important. For example, it applies whenever someone or something follows, records, or communicates a repeatable procedure.

People use the Hierarchical Steps pattern naturally when they do, think about, or communicate about performing tasks that are not extremely trivial. It is not likely that these people are aware of this pattern, and certainly not by the name I have given it for MetaAutomation, but they still do it.

Within this context, for software quality, XML is an established and very powerful metalanguage and set of technologies that supports hierarchical structure.

2.1.4 Forces

- Automation must persist the details of how it drives and measures the SUT, with a structure that supports later reporting and analysis.
- Logs are a powerful tool from a different problem domain, but by nature each log statement is persisted in isolation from all the others, thereby losing important contextual information.
- Performance of non-shipping automation code is not as critical as with product code.
- XML is a powerful metalanguage with implementations on all major platforms, and perfect for representing a hierarchy.
- Hierarchical Steps are more complex to implement than log statements.

2.1.5 Solution

In non-shipping automation code, record in a hierarchy all the information of how code drives and measures the SUT. XML supports this well, as shown in running sample code (see Introduction), free for reuse or modification.

All the details of a check are now persisted and available with context information. They are in a form that is suitable to efficient and reliable automated analysis. For the business, the hierarchy presents the business-facing details at and near the root of the hierarchy as well as the technology-facing details at and near the leaf nodes of the hierarchy.

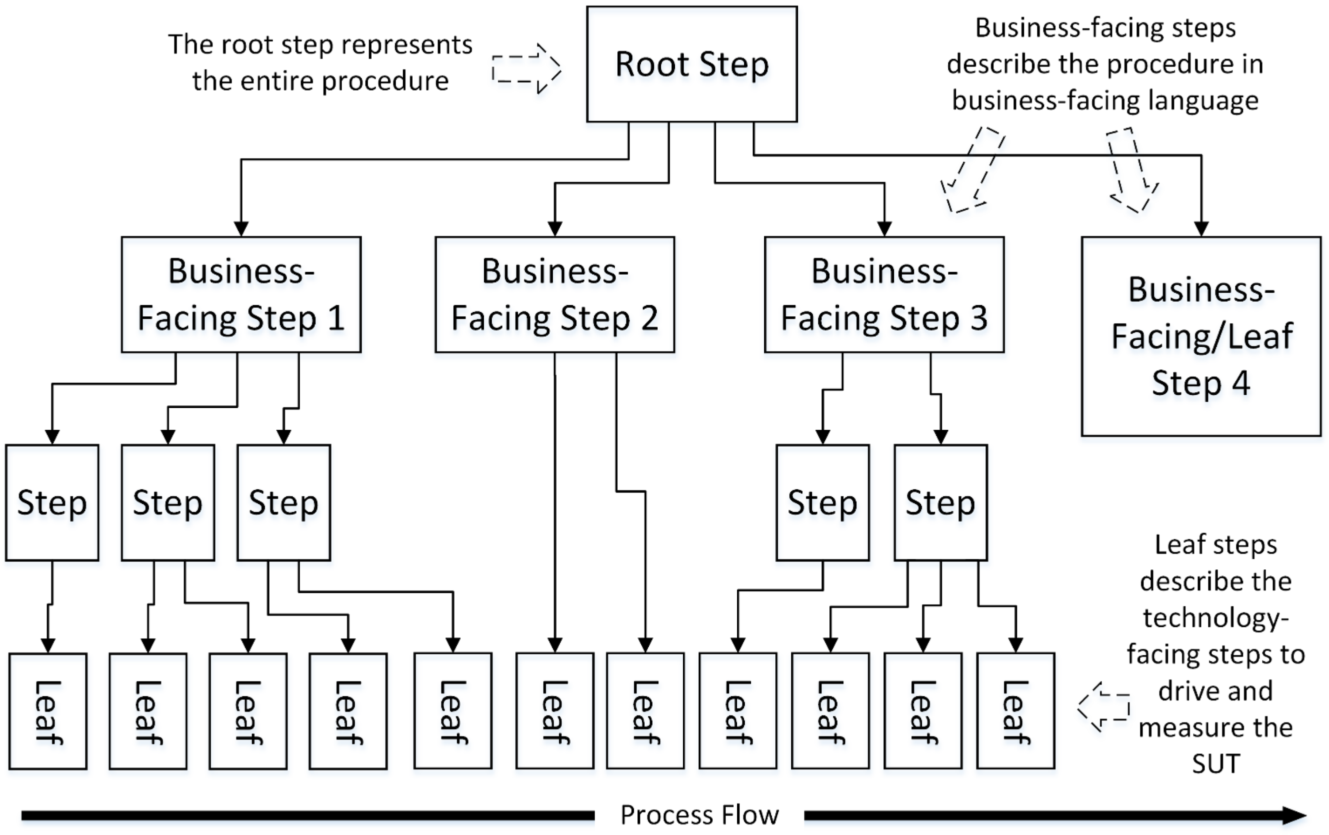

The root step represents the entire procedure
Root Step
Business-facing steps describe the procedure in business-facing language
Business-Facing Step 1
Business-Facing Step 2
Business-Facing Step 3
Business-Facing/Leaf Step 4
Step
Step
Step
Step
Step
Leaf
Leaf
Leaf
Leaf
Leaf
Leaf
Leaf
Leaf
Leaf
Leaf
Leaf
Leaf steps describe the technology-facing steps to drive and measure the SUT
Process Flow

Fig. 5. The hierarchy of steps for a procedure.

Implementing code to create a hierarchy at runtime is challenging, but the samples do it for you with open source (see introduction).

2.1.6 Resulting Context

Expression of a procedure is complete, extensible with more detail, changeable with limited impact, and very descriptive for failures.

All these benefits carry over to driving the SUT with automation. With the self-documenting automation code shown in the samples, this gives unprecedented clarity into what the automation code is doing [Collins and Lucena Jr. 2012].

The artifact of a check includes every detail. Every step has an explicit relationship to other steps, including peer steps, steps at a higher level of abstraction (towards the business) and lower levels of abstraction (towards the technology, i.e., driving the product). Also, every step in the check run artifact shows milliseconds to completion.

Since the artifact of the run of a check is in XML, it supports robust and performant analysis.

These aspects support the Queryable Quality pattern of MetaAutomation:

- The business-facing orientation of the root node of the hierarchy
- The technology-facing process of drilling down to the leaf nodes of the SUT-driving steps and measurements

A check failure propagates up the hierarchy from the leaf step where the error occurred to the root step, where the step name describes the entire check. If the same check has run before successfully, the record from the earlier check run shows all the steps, so it is clear which steps were blocked. Blocked steps record quality risk, because they show quality that might be unknown due to the earlier failure.

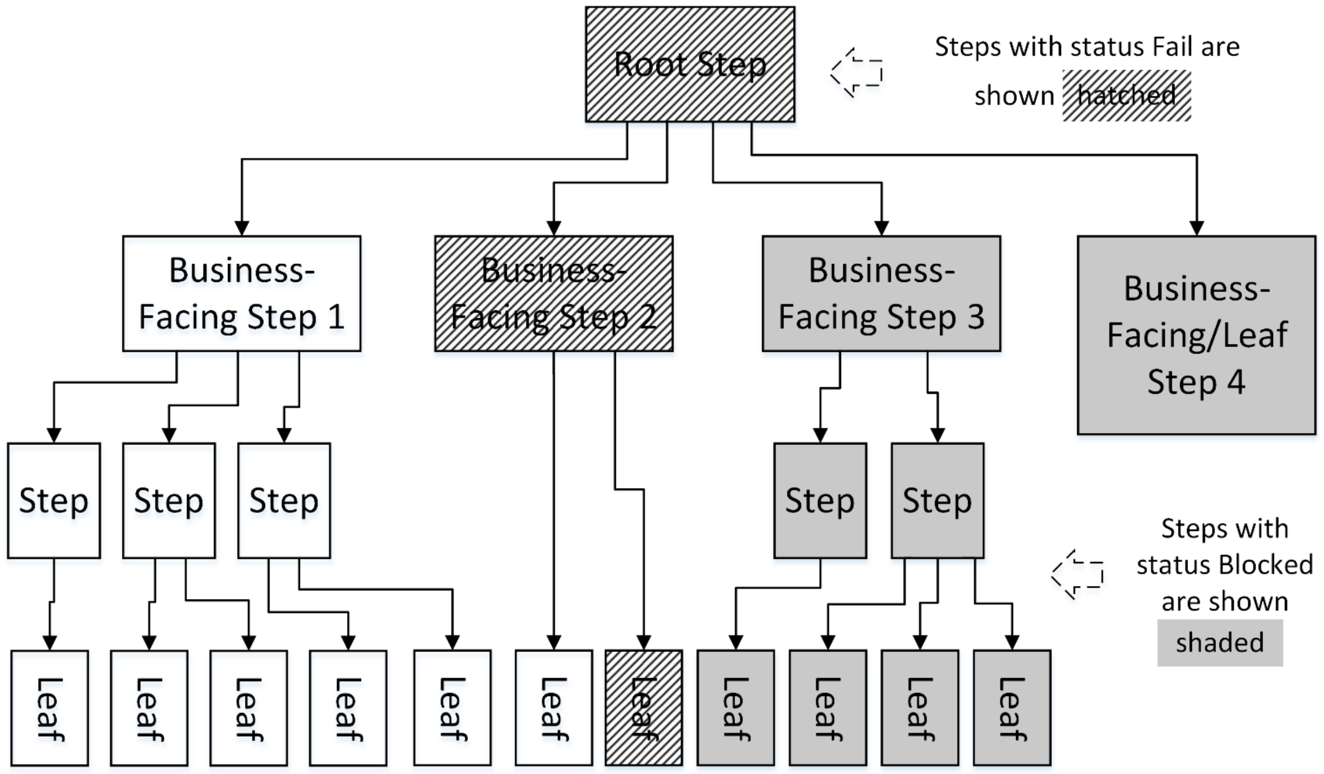

Root Step
Steps with status Fail are shown hatched
Business-Facing Step 1
Business-Facing Step 2
Business-Facing Step 3
Business-Facing/Leaf Step 4
Step
Step
Step
Step
Step
Steps with status Blocked are shown shaded
Leaf

Fig. 6. The hierarchy of steps for a procedure, with a failed step.

Adding or removing steps has a limited effect on the hierarchy, and therefore a limited effect on existing data. This in turn improves the value of check data over time for extended analysis.

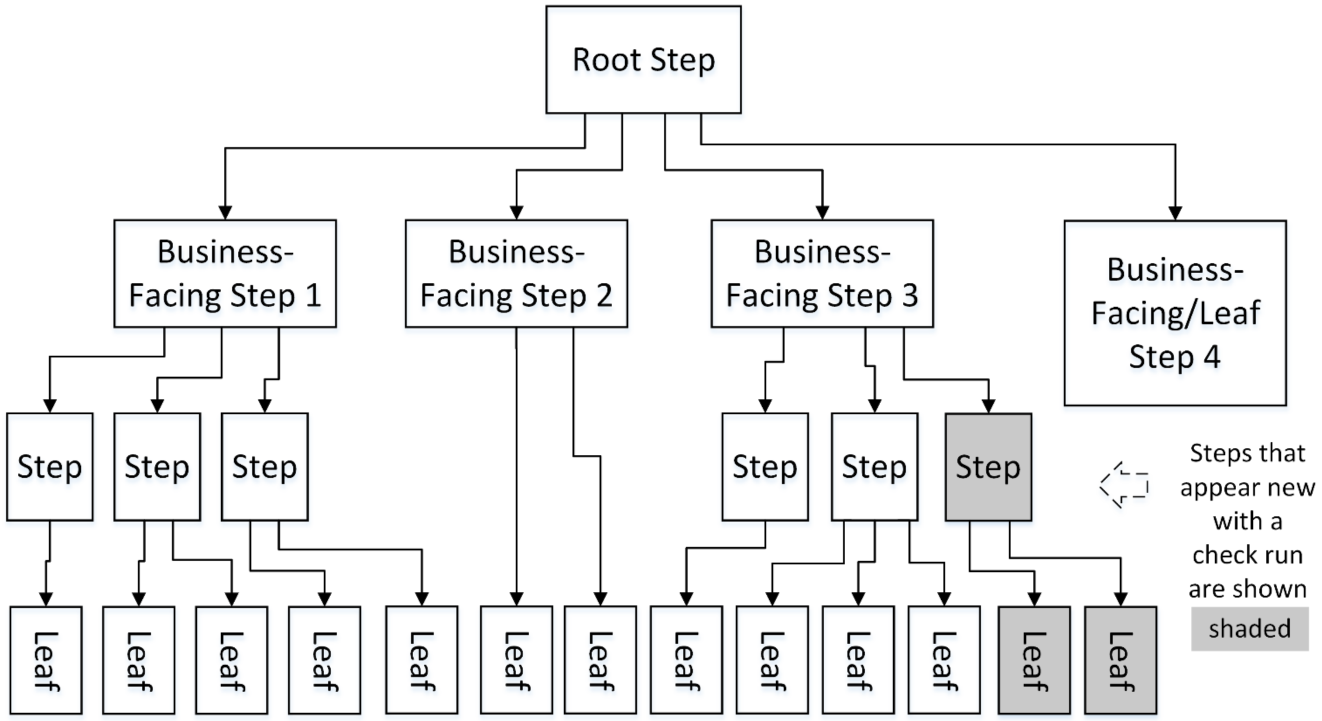


Fig. 7. The hierarchy of steps for a procedure, with added steps.

This detailed knowledge of the root cause of failure for a check support the Smart Retry and Automated Triage patterns. For example, as shown in Figure 6, an error that fails the check propagates up through the hierarchy to the root node. All levels of the hierarchy show, in their respective contexts, where the check failed.

2.1.7 Examples

A hierarchy of steps is a natural way to express a procedure to install a dishwasher in your home. If “Installing the Dishwasher” were the root node of a hierarchy, the child nodes would include “Remove the old one and clean the space,” “supply electrical power,” “hook up water supply,” “hook up drain,” and “test dishwasher and installation.” Each of those nodes has many more details as well, which, with a hierarchy, can go into child nodes. This has the advantages of making the installation instructions easier to follow and work-arounds easier to find should some part of the procedure fail or be inapplicable to the specific installation, as compared to the case of the procedure expressed as a (potentially very long) linear list of steps.

A hierarchy of steps expresses a complex recipe, for similar reasons.

Suzanne Sebillotte describes how a hierarchy is a best expression of steps to achieve a task, as this graphic from her paper shows [Sebillotte 1988] (Figure 8):

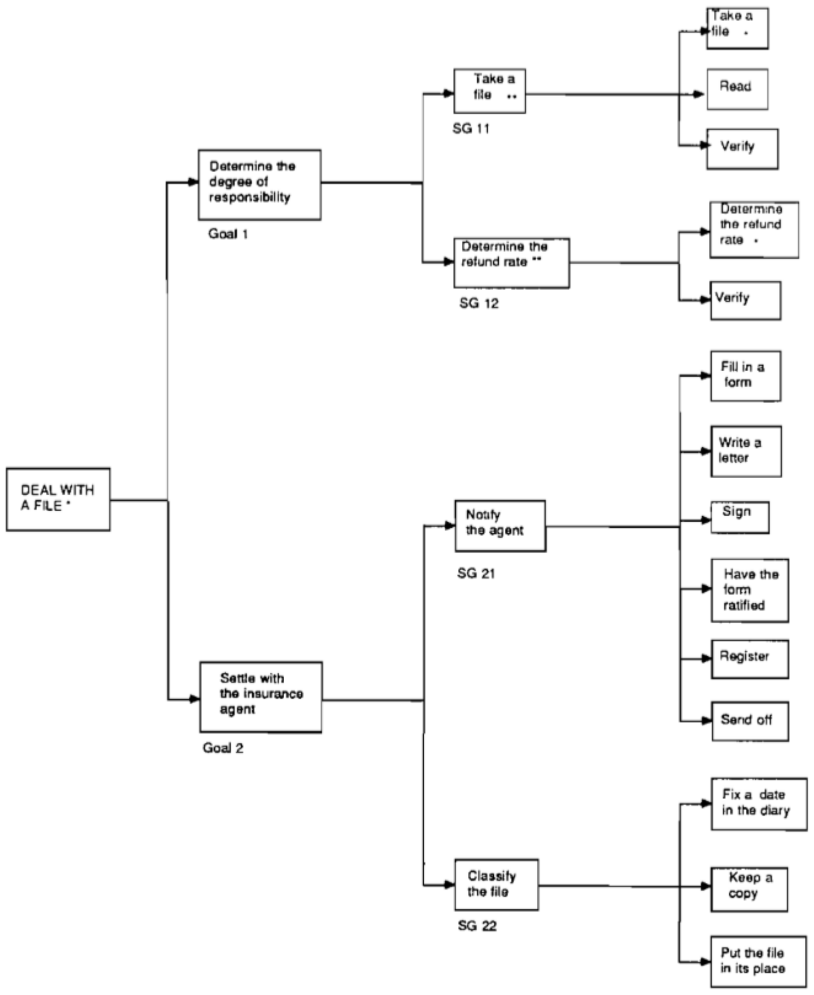
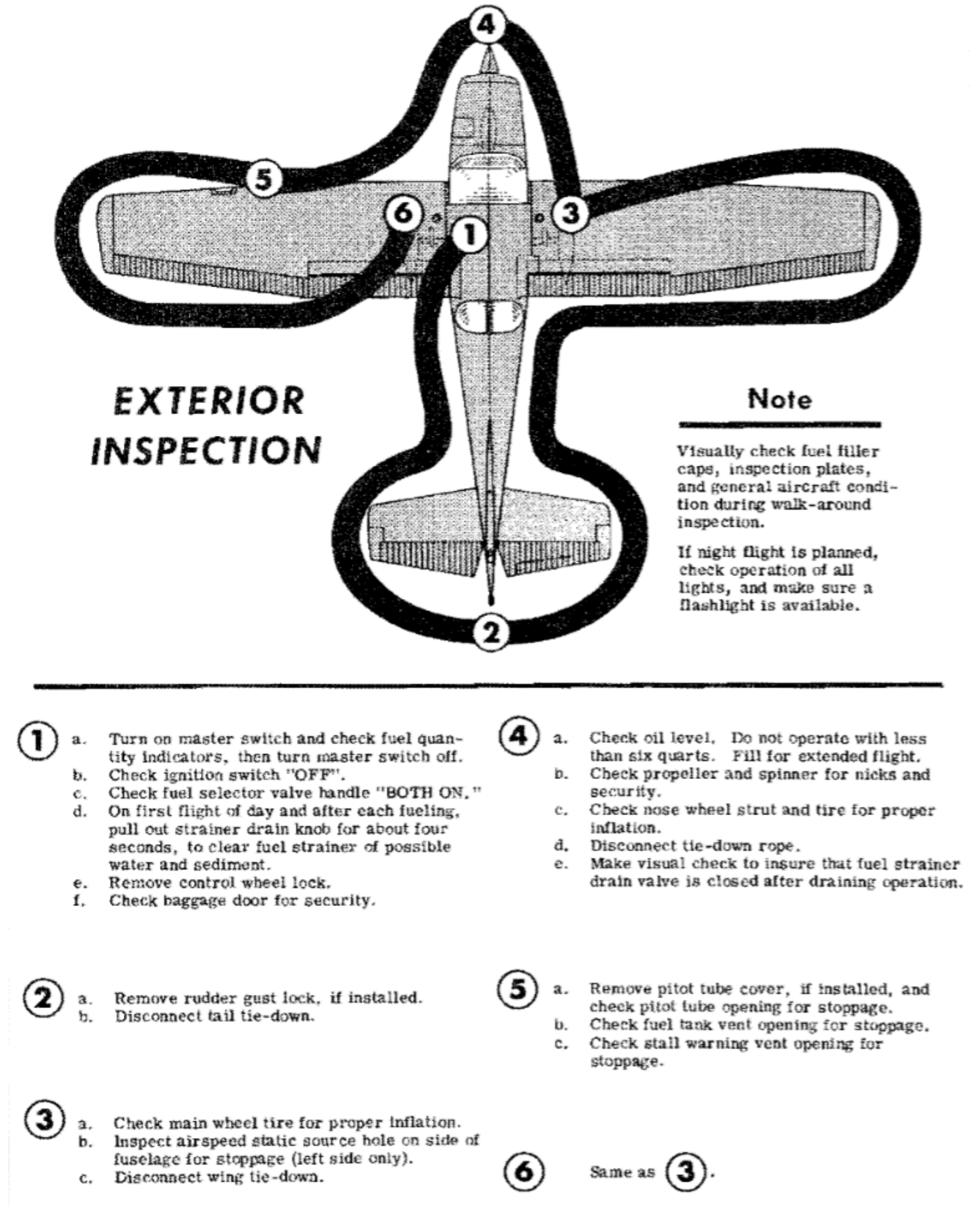


Fig. 8. Sebillotte's task hierarchy.

Hierarchical task analysis does not assume that the order of steps is always the same or completed sequentially. However, the Hierarchical Steps pattern describes procedures or parts of procedures where the order is stable, and either inherently sequential, or managed with thread synchronization to become sequential. This is necessary for quick and reliable quality measurements. The promise of MetaAutomation, to enable faster and more confident software development, depends on such measurements.

For example, given the GUI of an app, there are usually at least several different approaches towards achieving a given task. Each is measured with a separate check. Each such check would have a distinct but stable hierarchy.

Figure 9 shows a graphic from the owner's manual (graphic below) for a 1967 Cessna 172/Skyhawk. This is a nice visual example of how the Hierarchical Steps pattern occurs naturally, with the station steps (1-6) around the airplane standing for a higher level in the hierarchy and the lettered steps at each station standing for the lower level, i.e., child steps of the station steps [Cessna 1984].



EXTERIOR INSPECTION

Note

Visually check fuel filler caps, inspection plates, and general aircraft condition during walk-around inspection.

If night flight is planned, check operation of all lights, and make sure a flashlight is available.

(1)
a. Turn on master switch and check fuel quantity indicators, then turn master switch off.
b. Check ignition switch "OFF".
c. Check fuel selector valve handle "BOTH ON."
d. On first flight of day and after each fueling, pull out strainer drain knob for about four seconds, to clear fuel strainer of possible water and sediment.
e. Remove control wheel lock.
f. Check baggage door for security.

(2)
a. Remove rudder gust lock, if installed.
b. Disconnect tail tie-down.

(3)
a. Check main wheel tire for proper inflation.
b. Inspect airspeed static source hole on side of fuselage for stoppage (left side only).
c. Disconnect wing tie-down.

(4)
a. Check oil level. Do not operate with less than six quarts. Fill for extended flight.
b. Check propeller and spinner for nicks and security.
c. Check nose wheel strut and tire for proper inflation.
d. Disconnect tie-down rope.
e. Make visual check to insure that fuel strainer drain valve is closed after draining operation.

(5)
a. Remove pitot tube cover, if installed, and check pitot tube opening for stoppage.
b. Check fuel tank vent opening for stoppage.
c. Check stall warning vent opening for stoppage.

(6) Same as (3).

Fig. 9. Hierarchical Steps of preflight for a small airplane.

In the domain of running software, a good example is buying a plane ticket online. Web pages and controls on the pages arrange the steps of the ticket-buying experience. The itinerary request occurs on one page, where the traveler enters origin and destination, one-way vs round trip, leave and return dates etc. The leave and return dates include month, day, and year. This naturally forms a hierarchy: the root node is the overall task of buying a plane ticket. The children of the root are each of the web pages that hold information and choices for the passenger to make. Child nodes of the page nodes include items such as name and date. The name node includes first, middle, and last names. The date nodes include month, day, and year. This hierarchical arrangement leads the purchaser through a process where the context is clear for each step. This clarity and ease-of-use would be impossible if the steps were a long linear list of ordered items to read or enter information.

For the software *quality* problem domain, consider a hypothetical bank portal web app called "BankingAds." This web app enables a bank customer to make deposits, pay bills, withdraw from loans, make transfers between accounts and any other common banking operation, from any modern web browser. Advertisements appear to the side of the screen from an external advertising company. The differentiator for BankingAds is that (carefully anonymized) information from the end-user's account balance, activities, and history, feed the decision engine which leads to showing certain advertisements. The ads arrive asynchronously but may be based on what the end-user is doing at the time.

The team that creates and maintains BankingAds includes the QA or "Quality Assurance" role, that measures and supports quality and helps the team develop the software faster.

The Hierarchical Steps pattern, applied to automation, records in full detail driving and measuring the SUT. Each step shows context by position in the hierarchy and the record shows how many milliseconds each step took. In case of check fail, the result shows cause of failure at all hierarchy levels *and* all blocked steps.

This saves enormous amounts of debugging time, makes separate development of performance tests unnecessary, and communicates what BankingAds is doing correctly (or not) with performance information. Hierarchical Steps enables drill-down from the highest level of abstraction down to the atomic steps of driving the product, thereby making the information available to anyone on the team concerned with product quality.

The sample implementations (see Introduction) also show how hierarchical steps work with a very simple yet real-world software testing task.

2.2 Atomic Check

2.2.1 Summary

Atomic Check is about a simple, independent, focused automated procedure to verify a functional requirement and record detailed data on interacting with the SUT [Kappler 2016].

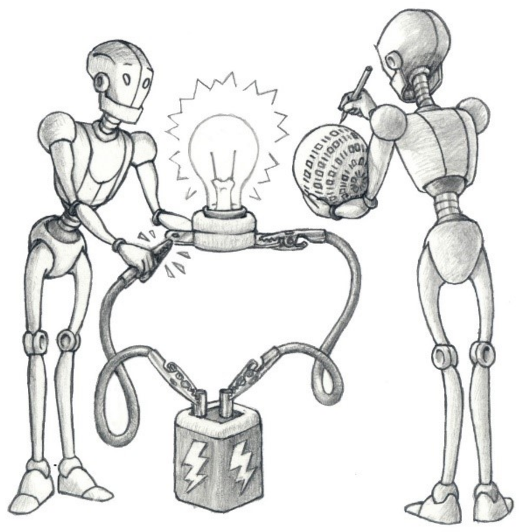


Fig. 10. Automation performing a check and recording detailed results.

2.2.2 Problem

With common practices, the business loses valuable quality data. This happens in many ways, including

- SUT Quality data is blocked (or, made untrustworthy) by earlier failures in a long and complex check.
- Quality data is too expensive because the checks take a very long time to run.
- In case of check pass, the team disregards what a check does to the SUT, in part due to the enduring misconception that test is only about finding bugs.

Risks to the software business include

- The checks become expensive on failure because the team needs a manual debugging session to find root cause, if they can reproduce the failure at all.
- If the team runs quality automation with stubbed-out or faked services that the SUT uses, that automation is not testing the entire SUT as end-users would experience it. Service changes or failures can be very high-risk events, and not verifying early and often how the SUT handles these creates significant unnecessary risk; there might be rude surprises at integration time.

How does one design a check to maximize the quality, quantity, and trustworthiness of data on the SUT, maximize the value of quality automation, and minimize quality risk?

2.2.3 Context

- The Hierarchical Steps pattern ensures complete data in a robust structure.
- Automation drives the SUT with repeatable checks.
- Automation makes measurements on the SUT and records all steps in a hierarchy.
- Automation persists all that data in the check result, along with pass/fail results.

2.2.4 Forces

- Computing power and resources used for the software development process, including quality management, are increasingly available and inexpensive.
- The team, especially software developers, must get detailed and trustworthy quality data quickly and reliably.
- A complex SUT needs many checks to verify that it meets functional requirements.
- For a check to be appropriately prioritized and informative, it must be traceable to functional requirements (and business requirements).

2.2.5 Solution

Design and implement checks as simply as possible, given that:

- Each one must have a single target verification or verification cluster, linked to functional requirements.
- Each must run with dependencies in place where possible.
- Each check must run completely independently of every other check.

For example, verifications other than the target verification or verification cluster should only happen if they will help the check fail faster and/or fail with more clarity on root cause.

To ensure reliable and complete data on product behavior, have the check implementation document itself with an implementation of the Hierarchical Steps pattern, e.g., as the samples do (see Introduction).

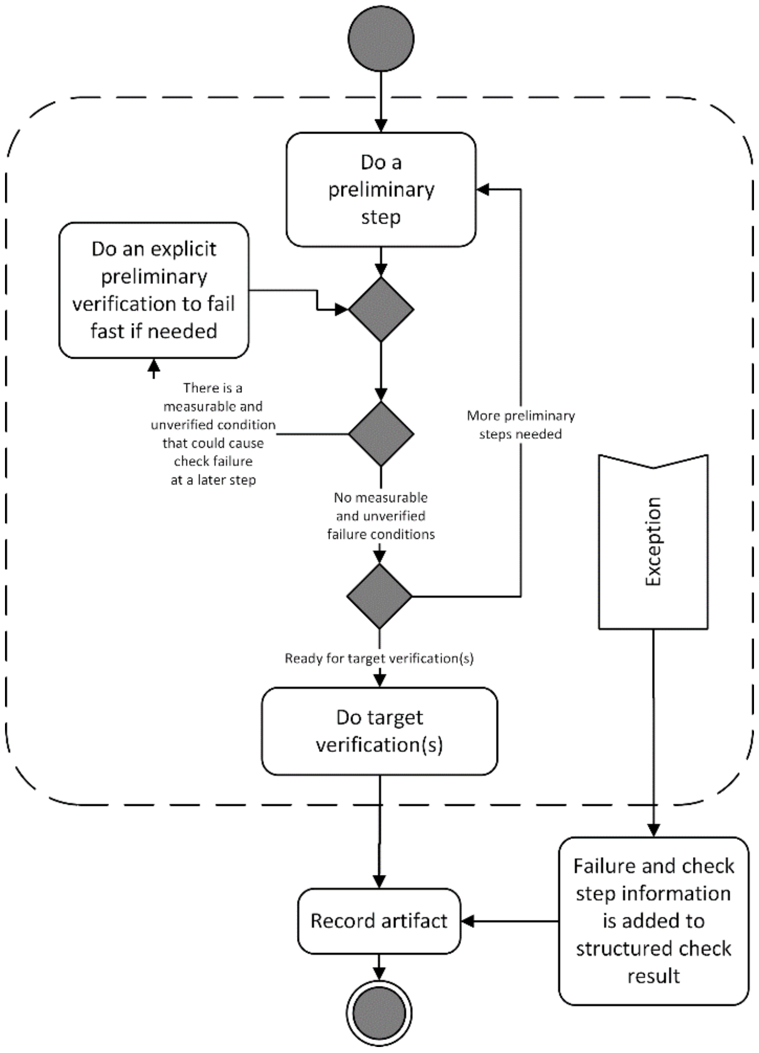

Do a preliminary step
Do an explicit preliminary verification to fail fast if needed
There is a measurable and unverified condition that could cause check failure at a later step
More preliminary steps needed
No measurable and unverified failure conditions
Exception
Ready for target verification(s)
Do target verification(s)
Failure and check step information is added to structured check result
Record artifact

Fig. 11. The Atomic Check UML activity diagram.

Include all dependencies of the software environment, including services, to manage quality risk. These are sometimes stubbed out for speed and reliability, but patterns of MetaAutomation take care of those issues. The best speed of a check run comes from applying the Atomic Check, Precondition Pool, and Parallel Run patterns. The Hierarchical Steps and Smart Retry patterns give reliability.

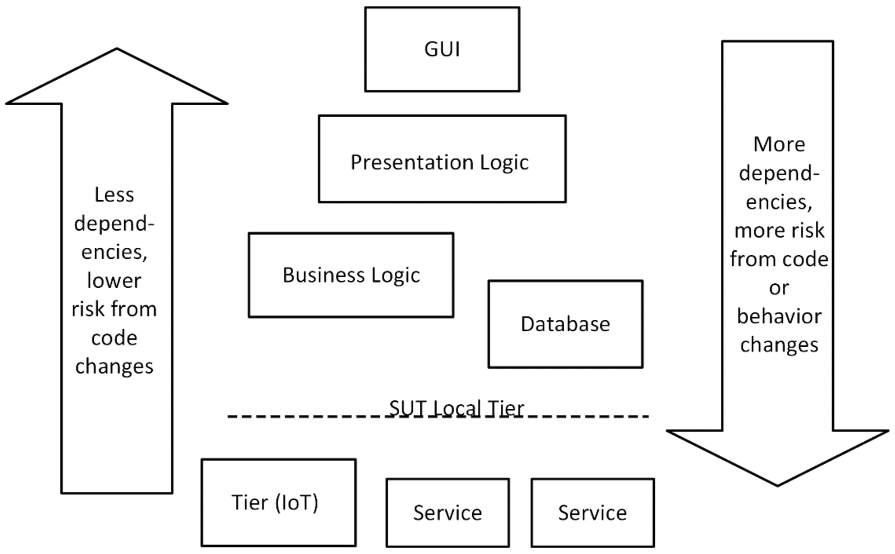

GUI
Presentation Logic
Business Logic
Database
SUT Local Tier
Tier (IoT)
Service
Service
Less depend-encies, lower risk from code changes
More depend-encies, more risk from code or behavior changes

Fig. 12. Dependency and quality risk.

Figure 12 shows the importance of including services and the less-dependent layers of the application for system testing.

2.2.6 Resulting Context

Simple verifications are faster and more trustworthy than complex ones, and give more pointed product quality data as well.

The Atomic Check pattern helps the team by describing a standard and a procedure for defining an optimal check. This pattern, although detailed, is flexible and general enough that it applies to a wide variety of software products showing deterministic behavior and the deterministic foundations of probabilistic systems (e.g., machine learning software).

Atomic Check supports Parallel Run because the individual checks are all independent of each other, and fast and simple.

Atomic Check supports Smart Retry and Automated Triage with fast, simple, scalable checks, each with just one target verification or verification cluster, and detailed data joining all levels of the hierarchy and pointing at root cause of failure

For Queryable Quality, all the check detail, for both passed and failed checks, is available. The business-facing root step of the check is central to check results, and drill-down is available to the leaf steps, that is, the technology-facing steps that drive and measure the SUT.

2.2.7 Examples

Effective practitioners in the field use simple, focused checks, per recommendations of Adam Goucher: [Goucher 2009]

- "…This rule is states that a test case should only be measuring one, and only one thing."
- "Test cases should not be dependent on other test cases."

As Meszaros writes, "We should avoid the temptation to test as much functionality as possible in a single Test… it is preferable to have many small Single-Condition Tests…" [Meszaros 2007, p. 359].

Returning to the hypothetical BankingAds example, Atomic Check makes the checks

- traceable to functional requirements
- as fast as possible
- as scalable as possible
- as simple as possible
- as trustworthy as possible

and, the structure provided by the Hierarchical Steps pattern enables a single check to run across multiple deployment tiers and/or layers of the application, cross-process or cross-machine as needed, to assure quality for the Internet of Things. Sample 3 (see Introduction) shows this.

2.3 Event-Driven Check

2.3.1 Summary

This pattern is useful when measured behaviors of the SUT respond to events that are either external or otherwise outside of the team's control.

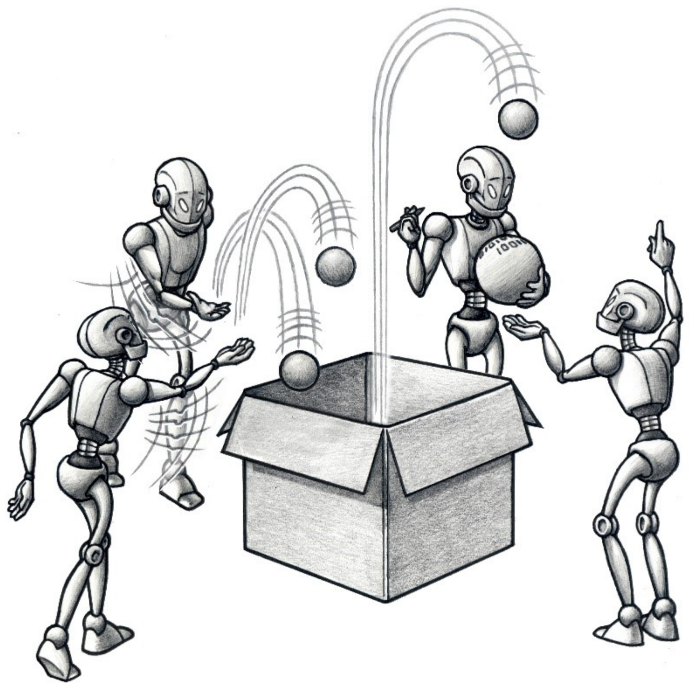


Fig. 13. With Event-Driven Check, automation supplies the events that drive the SUT, and the target verification may depend on an event as well.

2.3.2 Problem

If uncontrollable loosely-coupled external events drive part or all the SUT, conventional means of driving the SUT are not enough for deterministic measurements of functional quality.

2.3.3 Context

The SUT responds to external events, but functional quality measurements must happen in a repeatable and reliable way to support fast quality measurement, part of ensuring that quality always moves forward.

The Hierarchical Steps and Atomic Check patterns ensure good data and checks that are as simple as possible for the target verification(s).

2.3.4 Forces

- Some events are external and beyond the team's control, yet important to product behavior.
- Events subscribed to are much simpler than product dependencies on an external service.

2.3.5 Solution

Do as with Atomic Check, but for check runs, subscribe the product to events generated by automation, rather than the external events.

Drive the events as part of the check.

Measure SUT behavior directly if possible, but if not, subscribe to the correct events and block on them if needed.

2.3.6 Resulting Context

The resulting context is just like the resulting context of an Atomic Check instance.

2.3.7 Examples

Event-Driven Check pattern appears in GUI-driven checks, where events to and from the GUI interact with common automation tools.

This pattern works to automate decoupled systems joined only by event emitters and consumers.

2.4 Extension Check

2.4.1 Summary

Suppose the team needs quick, reliable verification of an important quality attribute of the SUT, but there is no way to drive the feature deterministically. An Extension Check avoids this problem by operating on data provided with an earlier check result.

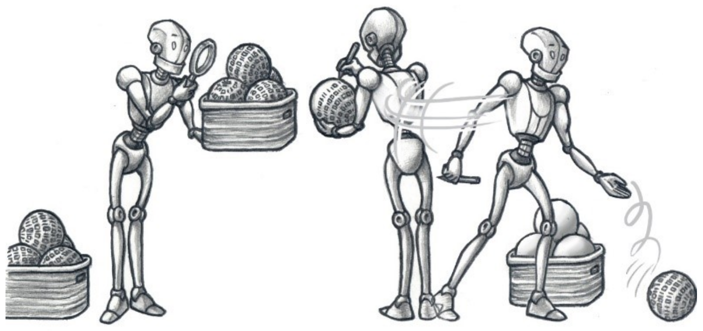


Fig. 14. Automation making checks on the artifacts of earlier checks for the Extension Check pattern.

2.4.2 Problem

The team needs to measure some behavioral aspect of the SUT quickly and reliably. Due to SUT architecture or environment, the behavior cannot be driven deterministically.

2.4.3 Context

Hierarchical Steps gives very detailed, structured, and trustworthy data on the SUT.

Some aspects of some systems cannot be driven deterministically.

During a check run, the SUT, instrumented as needed, might show important but non-drivable behaviors.

2.4.4 Forces

- There are important non-deterministic behavior aspects of the SUT.
- Repetitive manual tests are boring and expensive.
- Implementing this pattern involves some cost.
- The measurement and reporting on the quality aspect concerned must not interfere with higher-priority checks.

2.4.5 Solution

Add code to other checks to measure and add data in child nodes to enable the extension checks. Do not fail, slow or block those checks on the extra measurements. Extension checks can measure the non-deterministic quality criteria by reading the artifacts of those earlier check runs that include the software unit(s) where those quality criteria are at issue. The extension checks fail on criteria of the data alone, and only if the sought data is available in the artifacts of the earlier checks.

2.4.6 *Resulting Context*

The quality automation system can verify non-deterministic aspects of the SUT with fast, scalable checks.

2.4.7 *Examples*

There are many examples of human endeavors that involve analysis after-the-fact of data from uncontrollable experiments, for example:

- Significant sociology research, e.g., anthropology studies, is done without controlled experiments because people are difficult to control, especially because some types of control would be unethical.
- Most Astronomy research uses data from "experiments" in the remote universe that humans do not have the power to start or control.

For the BankingAds app, although the ads come from an external company, people on the project still must verify that the ads are correctly served in response to end-user activities, balances, etc. and of course asynchronously during other activities. The new-ad event gets a hook that adds information about the ad to the check as the check is running, so the timing, identity, type etc. of the ad shows up in the artifact of the check.

After the SUT-driving check run, an Extension Check does the analysis to decide if the ads are correct or not, with acceptable timing and other criteria.

2.5 Precondition Pool

2.5.1 *Summary*

If systems give prepared resources independently of the check itself, the check can run faster and more reliably, and create more focused quality data.

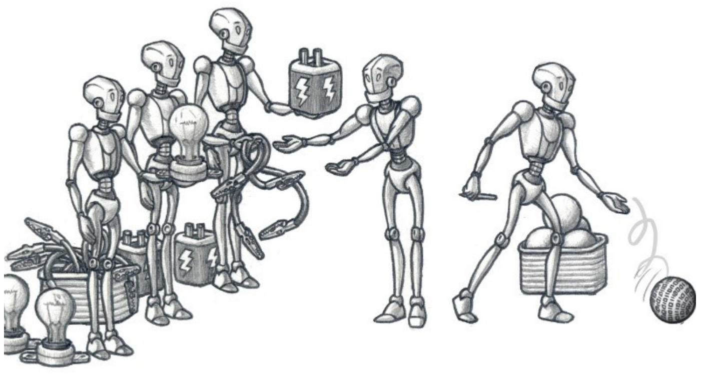


Fig. 15. Three Precondition Pool robots give ready resources for the check to run.

2.5.2 *Problem*

If every check runs with all preconditions created in-line with the check, the run will be slow and have more potential points of failure than needed for a given quality measurement. Over many check runs, more potential points of failure will dilute the value of the quality data.

2.5.3 *Context*

Checks focus on target verifications. Verifications need resources that may be runtime-independent of the target.

For example, every check runs in an environment.

Depending on the SUT, quality automation can break out other resources as well. For example, user accounts, documents, database images, externally-facing accounts, may all be part of a check but runtime-independent of the target verification or verifications.

2.5.4 Forces

- The checks run faster and more reliably when preconditions are managed and queued up out-of-line.
- The checks are simpler when preconditions are handled separately, so can better focus data from the Hierarchical Steps pattern.
- Moving preconditions out-of-line and out-of-process from the check can add implementation complexity and overhead to the overall quality automation solution.

2.5.5 Solution

For your SUT, consider which resources a pool can manage out-of-process to the check itself. You may already be managing automation environments, with automated choice of environments; if so, you are using the Precondition Pool pattern already.

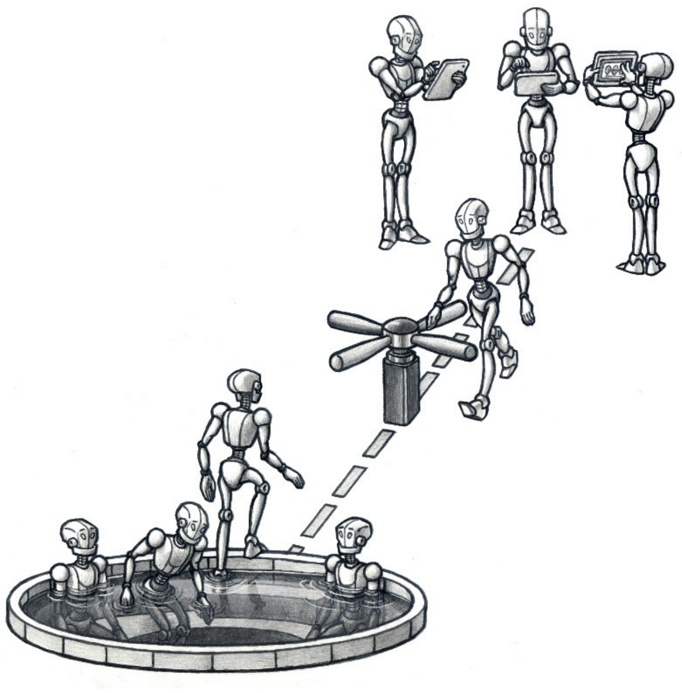


Fig. 16. Precondition Pool resource flow for an object type.

For each resource type, manage it as an independent pool, completely out-of-process relative to the check run or runs. Quality automation checks out pool items as needed for use with a check, then checks them back in. The pool implementation restores them as necessary to the desired state. The pool keeps enough such items that they enable the check run to run at optimal speed. Even if the check run uses so many pooled resources that the pool takes significant maintenance overhead during the run, there is little added overhead compared to the case where the resources are not pooled. The pool can also expand to handle more such resources and keep them at the ready, so when a check run is launched, it completes sooner. The checks themselves are more reliable and simpler, so create better and more useful data on the SUT. The entire check run scales better with resources, too.

Precondition Pool instances can also exist for sub-types of resources. For example, there can be a Precondition Pool instance for each role of a test user, for an application that uses role-based security.

2.5.6 Resulting Context

By helping the checks run quicker and with more focused quality data, Precondition Pool helps the Parallel Run and Smart retry patterns run more effectively, and it reduces the duration of the set of checks for a given run.

If a given Precondition Pool takes significant computing resources, then it might be desirable to cache many those resources up front so that when a check run begins, those resources are available to speed the checks and deliver results faster.

With the preconditions taken care of separately, the checks are simpler and the data that results from running the checks is therefore easier and more valuable to work with.

With the pools handling errors associated with managing the preconditions, the checks have fewer points of potential failure and so are more reliable.

2.5.7 Examples

The Setup and Teardown phases of the Four-Phase Test pattern named by Meszaros are precursors to Precondition Pool because the intent of setup and teardown is of a "fixture," i.e., something that needs to happen for the test, but which is not the target of the test. That so many people used the pattern is therefore an example of managing check resources as preconditions. With Four-Phase Test, however, the phases always happen in-line, which has negative consequences on check complexity, reliability, performance, scale, and the quality of the artifact data.

The Precondition Pool pattern applies where checks run across different machines or virtual machines; a pool implementation in this case manages the available environments where the SUT (or, the client part of it) is configured and running.

Other examples of external resources that can be managed with this solution include:

- thread pools
- user accounts of diverse types and states
- internal or external databases of test data or standard product configurations
- documents of certain states or storage locations

etc.

Discovering and characterizing the pattern creates opportunities for the team. If the system gets more Precondition Pool instances to manage more resources, the checks will be more scalable, faster, simpler, and more effective.

Returning to the BankingAds app example, Precondition Pool applies to managing the many environments in which the checks can run scalably, as well as customer accounts of diverse types and balances. Having pools to handle user accounts makes the checks simpler, faster, and more scalable, and thereby shortens the overall check run and improves the quality of data from the check run.

2.6 Parallel Run

2.6.1 Summary

If checks run in parallel, they scale with available computing resources.

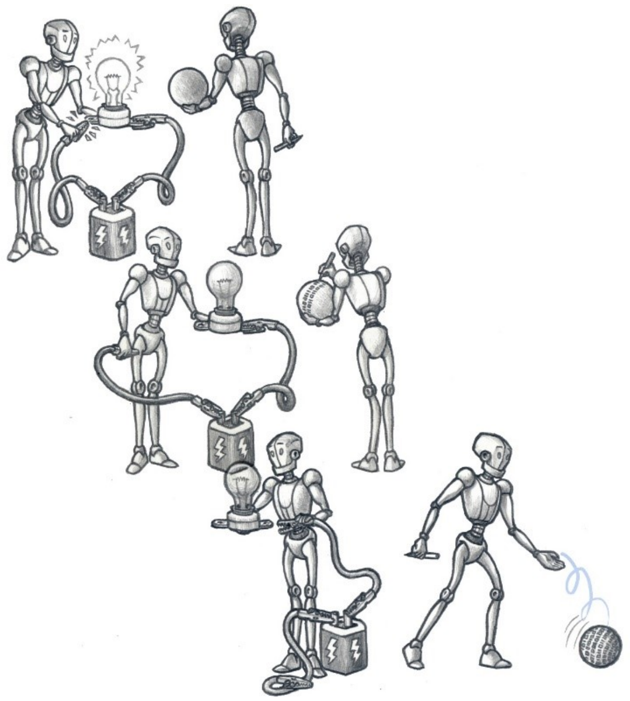


Fig. 17. Three instances of automation processes running checks in parallel.

2.6.2 Problem

If checks run sequentially, for a large set of checks, the entire check run can take a long time. The business needs to run many checks quickly.

2.6.3 Context

The Atomic Check and Precondition Pool patterns ensure that checks are optimally fast and run independently of each other.

Precondition Pool supplies preconditions for the checks to run, speeding and simplifying the checks.

Computing resources are increasingly available at decreasing costs.

2.6.4 Forces

- The team needs a set of checks to run as fast as possible.
- The number of checks to run might be very large.
- The checks can run independently of each other.
- There is some development cost to implement a Precondition Pool.
- There is some runtime cost and network overhead to interface with it, to enable the checks to run in parallel.

2.6.5 Solution

With a Precondition Pool implementation handling many machines and environments, run the checks in parallel [Caspar et al. 2014] [Garg and Datta 2013].

Without parallelization, the checks all must run sequentially, as the UML sequence diagram Figure 18 shows:

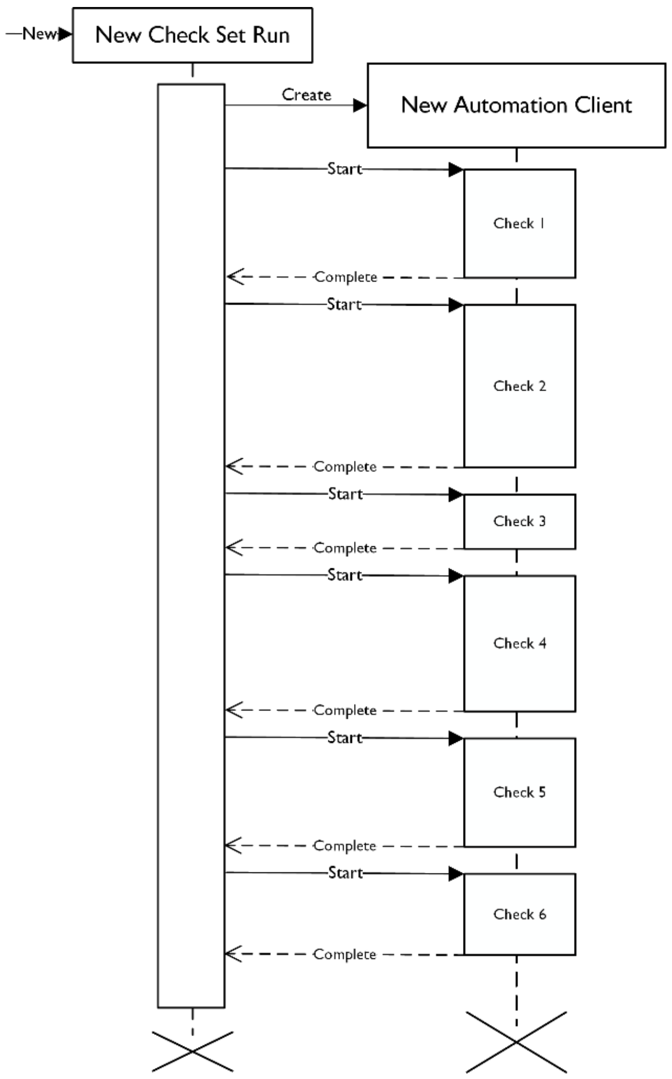


Fig. 18. Checks running sequentially.

With parallelization, the checks can run on an arbitrary number of clients, so they run faster. Given a large number of clients, the speed at which the checks run is almost arbitrarily fast. Figure 19 shows the speed increase with the same checks run across three clients.

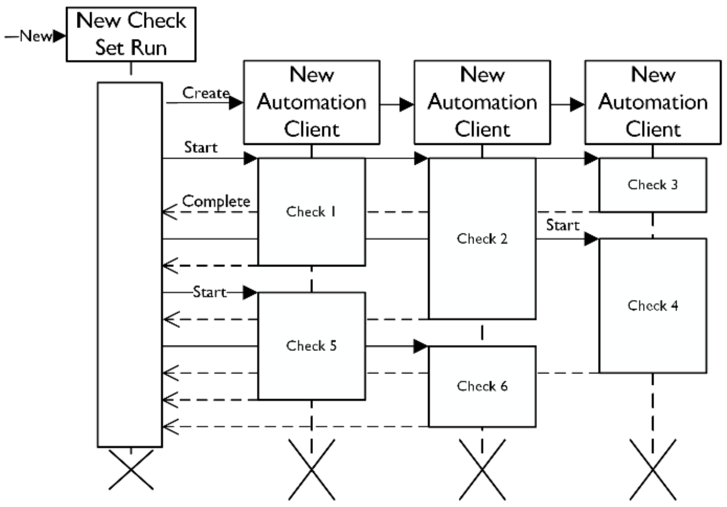


Fig. 19. Checks running in parallel.

2.6.6 Resulting Context

Since they can scale with computing resources, the checks can run faster overall, and potentially much faster, making it possible to run a much larger number of checks and get much more data on the SUT.

Smart Retry is now effective because a retry of a given check will happen quickly with minimal impact on the overall check run.

Queryable Quality is more effective because there is more data on the product.

2.6.7 Examples

This pattern appears in, e.g., web servers for high-volume sites where requests are handled on different threads, cores, processors, and/or machines. Weather simulations also depend on massively parallel processing.

For the BankingAds example, the number of checks is quite large, and the app is very impactful to people so functional quality is very important. The Atomic Check pattern enables each check to run independently, so there is an opportunity as well as an incentive to run the checks in parallel across an arbitrarily large number of virtual machines. The check run can therefore be very fast, and integrate into the development process with manageable disruption.

2.7 Smart Retry

2.7.1 Summary

It is a common pattern to retry a check on fail, but without much data on what the check is doing or where it failed, only a "dumb" retry is possible. Root cause of a failure, or whether a failure is reproduced, is unknown to the automation. The opportunity cost of not having the data is that clarity is only possible through manual follow-up [Jiang et al. 2017].

With detailed and trustworthy runtime data on how automation drives the SUT, quality automation can include knowledge of root cause and failure persistence in automated decisions. This delivers more value for the team, and more quickly.

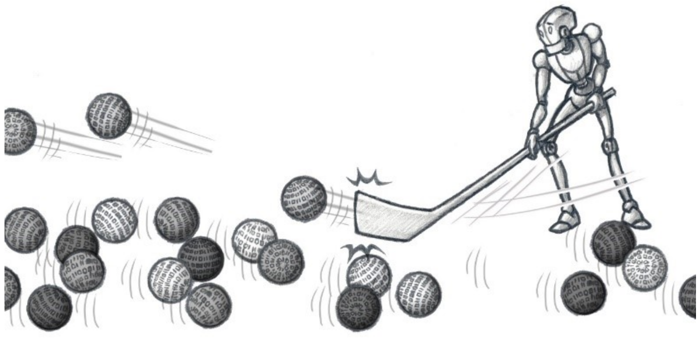


Fig. 20. Automation choosing checks to retry, based on detailed data from a failed check.

2.7.2 Problem

With conventional practices, on a team that uses automation for quality, false positives may be a significant business cost as well as an annoyance. They tend to block informative quality measurements and decrease trust in check results, therefore interfering with the value of the quality automation.

Some check failures might need to be reproduced for clarity, more data or just the trust that it is not a one-off failure. If they cannot be reproduced, that is valuable information too; but with conventional approaches none of that data is available to or from quality automation.

2.7.3 Context

The Hierarchical Steps pattern gives detailed and highly trustworthy data, including showing root cause of a given check failure.

The Atomic Check pattern ensures that the check is as simple as possible, given the target, so the data is focused.

The Atomic Check and Parallel Run patterns make it possible to run many checks and retry any failed check quickly.

Sometimes the checks fail.

2.7.4 Forces

- A check that fails due to an external resource, but would probably succeed on retry, might be a false positive and not actionable by the team.
- Interrupting team members with false positives is a significant cost to the business.
- If a given failure can be reproduced before presenting it to the business, that increases trust in and actionability of the failure.
- A failure that cannot be reproduced is not necessarily a false positive; depending on root cause, it might be a sign of a bug in the SUT.
- In some systems, e.g., avionics, any failure may be significant.
- Implementing the pattern and configuring it for the SUT may have significant cost.

2.7.5 Solution

The data in each check failure shows the root cause with detail on driving the SUT. Configure the quality automation system to either run the check again in every case of failure, or run it again only if the root cause is not in the SUT or otherwise not actionable.

Figure 21 shows a simple implementation of Smart Retry that assumes that all check failures are candidates for retries, no matter the root cause of the check failure:

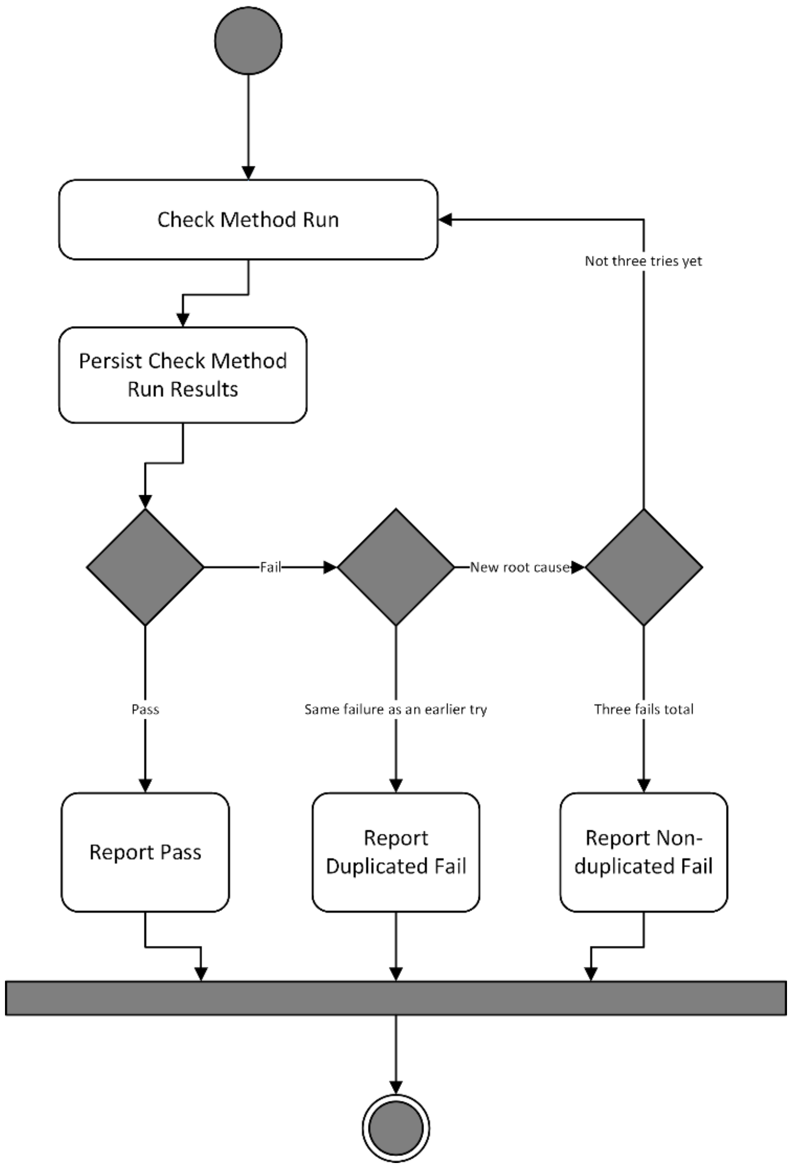


Fig. 21. The Smart Retry activity, in the case where all checks may, in case of failure, be candidates for retry.

A slightly more complex implementation does not always retry. For some failure cases in some SUTs, retry is not desired because the failure is always important and actionable. The detailed artifact from a failed check has the data to decide whether this is the case. For example, a non-deterministic failure due to SUT code might be actionable, and not a candidate for retry.

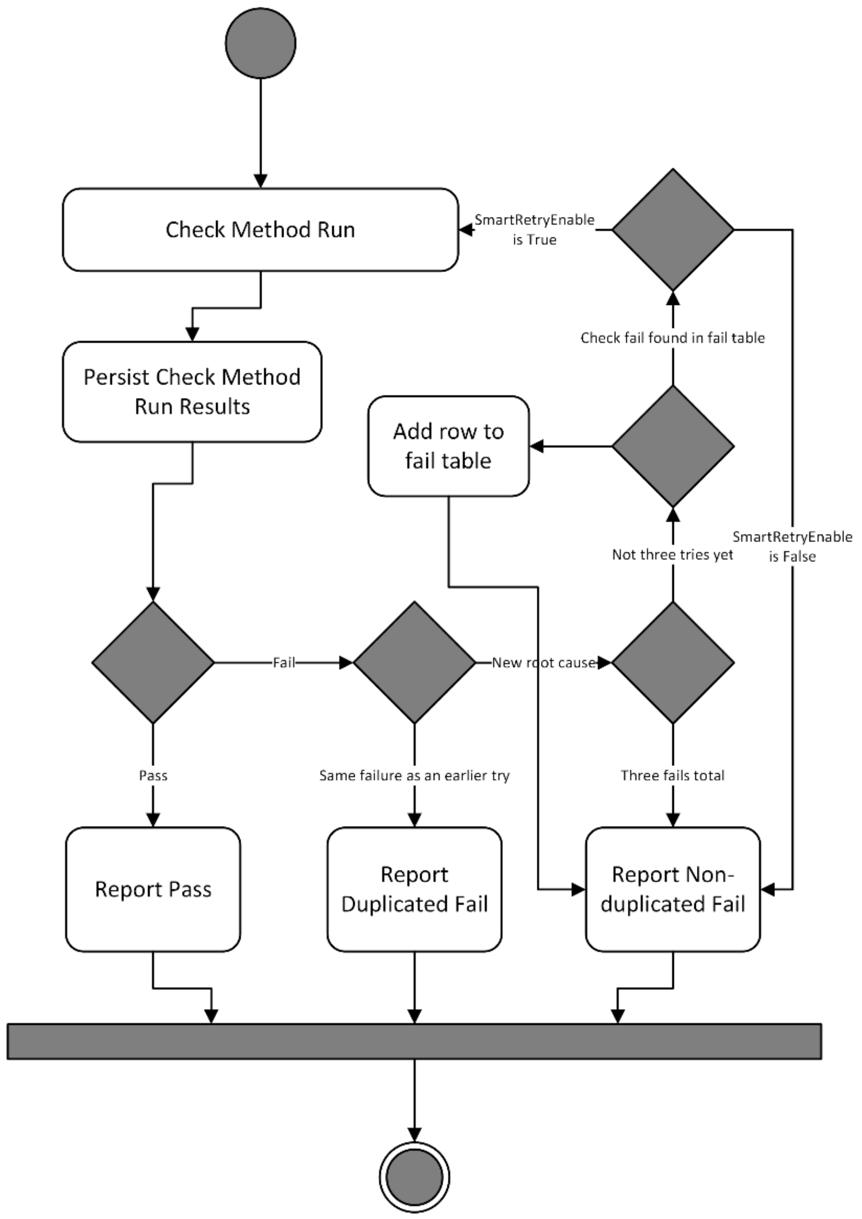


Fig. 22. The Smart Retry activity, where the engine does not retry by default.

Figure 22 shows the case where, given some failures, no retry happens. A table of root cause information with the manually enabled SmartRetryEnable flag enabling retry, decides whether there is a retry. This could also work with a quick data-driven set of logic tests.

Smart Retry bundles the information from retried checks before the data is available to the rest of the quality automation system. Here are four potential cases, and how Smart Retry might handle them:

1. For checks that failed, but the immediate retry passed, Smart Retry bundles data from both check attempts and marks the bundle as a pass.
2. For checks that failed, when the immediate retry reproduced the error, data from both check attempts is bundled as a fail. Since the failure reproduced automatically, it has improved value and is more actionable for the business than if it were just an isolated failure.
3. For checks that fail and map to a retry, but then fails again in a way that does not map to a retry, the resulting bundle is a failure with the non-retry check fail given higher priority because it is an action item for a developer.
4. For checks that fail, and then fail again with different root cause that also maps to enable a retry, and the third try fails to reproduce either of the first two failures, data from all three check attempts is bundled and shown as

a non-reproduced fail. If one of the failures does not map to enable a retry, it is marked or ordered at higher priority for the rest of the quality automation system.

2.7.6 Resulting Context

The Automated Triage implementation will not receive false positives, so the team will not get any such unnecessary distractions. See above in the solution for this pattern (section 2.7.5) which failures are actionable, and therefore worthy of a targeted notification.

Queryable Quality shows all check results, including checks that passed the first time and checks that experienced failures. For retried checks, the resulting bundle of check tries is displayed the same way as a non-retried check is, with a pass or fail status depending on the status of the bundle. Drill-down exposes the multiple tries of the check.

2.7.7 Examples

Microsoft, Google, and others use a pattern called "Retry" which is to simply retry a check on failure, up to three tries total. This is useful for automation on, e.g., a graphical user interface (GUI) or web browser and where the synchronization points are inaccessible, not available at all, for some reason too difficult or expensive to access, or they time out sometimes anyway. This is a much simpler version of the pattern presented here.

However, applying the simpler retry pattern is risky: it makes no attempt to distinguish between failures due to unavoidable race conditions in a GUI or actionable (and, potentially fixable) race conditions in the SUT. An actionable failure could be hidden by a retry that passes, and never discovered because (in the absence of the Hierarchical Steps pattern) the data is not there.

In the Office team at Microsoft, the complexity of the SUT is such that it includes nondeterministic business logic conditions (see the Extension Check pattern of section 2.4). The team uses the Retry pattern here [Roseberry 2017].

Unlike the Smart Retry pattern, however, Retry has no capability for deciding at check run time whether a specific failure cause is reproduced or whether, based on root cause of failure, the retry should happen in the first place.

For the BankingAds example, given that the end-user interface for the app is a web browser, there may be automation failures due to race conditions in the client interface. These are candidates for retry according to Smart Retry. If the timeout failure is reproduced at the same leaf step for a check, then the failure becomes actionable by either someone in the QA role (who might increase a timeout, depending on other timing information for the check) or a developer who works with the interface.

For checks that discover an incorrect monetary balance, no retry is in order; that failure is actionable.

2.8 Automated Triage

2.8.1 Summary

Given that checks have detailed, trustworthy data on the SUT, and failed checks know about root cause of failure, Automated Triage shows how to send targeted notifications as needed.

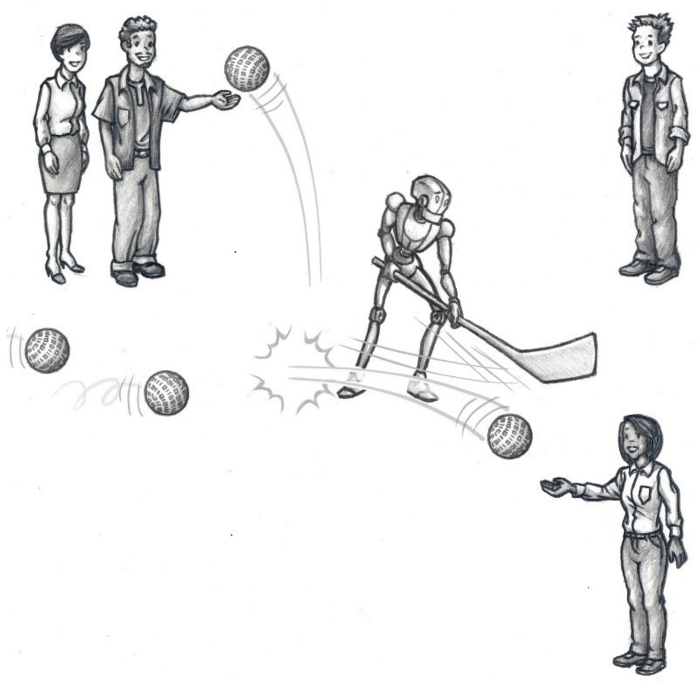


Fig. 23. Based on check data, automation directs notifications.

2.8.2 *Problem*

With conventional quality automation, all check failures trigger notifications, or are part of notifications, sent to a sizable portion of all people on the team. It then becomes a manual task to decide who, if anybody, is the best person to take an action item based on a given check failure. This is an unnecessary distraction and cost to the business.

The business needs targeted push notifications, but how to send them only to those who need to know?

2.8.3 *Context*

Hierarchical Steps gives very detailed, structured, and trustworthy data on every check run, including cause of failure at all hierarchy levels for those checks that fail.

Atomic Check gives checks that are as simple as possible, given the target verification or verification cluster. The detailed data in the targeted check shows who gets the notification.

Smart Retry can show whether a given check failure is a false positive or, in case of retry, whether the exact failure can be reproduced.

2.8.4 *Forces*

- There are business costs when team members receive notifications that are not actionable; the team gets distracted, it decreases trust in the quality automation system, and as a result, decreases attention to the *important* quality-related notifications.
- There is overhead to implementing and configuring the Automated Triage pattern.

2.8.5 *Solution*

Use a rules engine, configured with criteria to choose based on data stored form the check failure, to send notification(s) only to targeted people or a small discussion list.

2.8.6 *Resulting Context*

The notifications go to those who would act on them, or to a discussion list in case there is a small set of people, one of whom must act on it.

Notifications sent might include basic data on the failure, including

- whether the quality automation system has reproduced the failure

- whether this failure exists and is correlated with a known bug
- a link to a Queryable Quality implementation view of the failure

etc.

The notifications may also include links to an implementation of Queryable Quality, so recipients can investigate further.

Receiving only targeted messages from the quality automation system means there are many fewer of them in recipients' inboxes, and these messages are far more likely to get the right priority. This is important to keep SUT behavioral quality always moving forward.

2.8.7 Examples

A simpler and more modest system for directing notifications is in use in the Office team at Microsoft [Roseberry 2017]. The available artifact data is much more modest than what MetaAutomation enables, so options for directing the notifications are correspondingly simpler.

For the BankingAds app, Automated Triage directs actionable communications to specific people or groups. For example, if a web page object timeout was reproduced by the Smart Retry pattern, a notification goes to a person or list in the QA role. In case of an incorrect bank balance, a correctly configured Smart Retry implementation would not do a retry, and a notification would go directly to a developer or group.

2.9 Queryable Quality

2.9.1 Summary

Queryable Quality provides rich access to quality data on the SUT, for anybody on the team concerned with quality, including QA, test, the dev team, etc. Team members can, if they wish, drill down from a brief description of the check, to the business-facing steps, all the way to the technology-facing steps and measurements on the SUT.

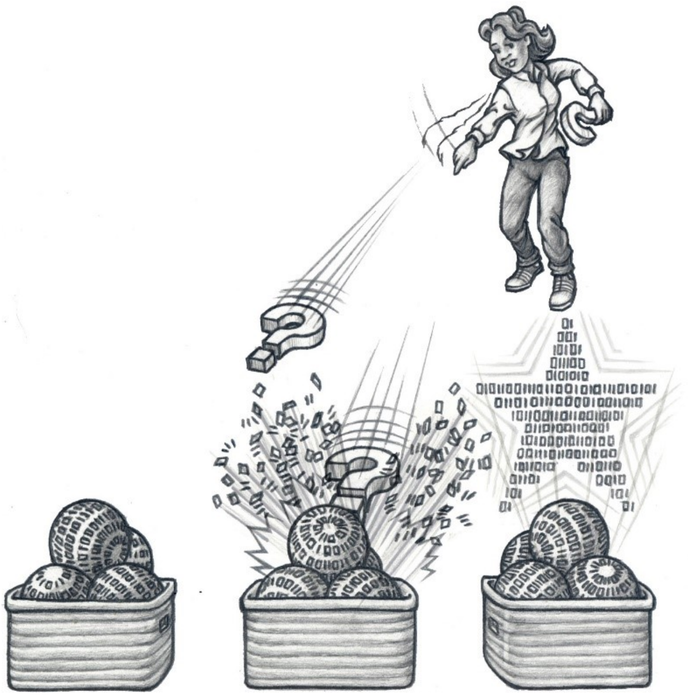


Fig. 24. A team member querying the data for the Queryable Quality pattern.

2.9.2 Problem

The business needs role-appropriate transparency into SUT functional and performance quality, so anybody on the team can access what is going on with the product, in the correct detail level and in a meaningful way.

2.9.3 Context

The Hierarchical Steps pattern, Atomic Check, Precondition Pool, and Parallel Run give vast amounts of highly structured and trustworthy data on product quality.

Smart Retry ensures that non-actionable failures do not interrupt people's work flow.

Automated Triage includes links to Queryable Quality with sent notifications.

Team members will want access to the detailed and trustworthy data on the SUT for performance and functionality.

2.9.4 Forces

- Quality data must be easily viewable, link-able and queryable by team members.
- The data generated by the quality automation must be persisted and served with read-only access for the lifetime of the SUT.
- Implementing and supporting a Queryable Quality portal (or, quality portal) has cost.

2.9.5 Solution

Implement and deploy an interactive portal, client or other human-computer interface that is internal to the company. The portal enables rich and configurable query, display, drill-down and export capability.

With drill-down capability, while the business-facing view of a check is at and near the displayed root node of the hierarchy, drill-down is available to show more detail as desired. If they wish, team members can drill all the way to the atomic technology-facing steps and measurements of the SUT. Operations and measurements on the SUT are completely self-documenting, and Queryable Quality surfaces this information to the team for casual perusal and deep analysis.

2.9.6 Resulting Context

All data on the SUT for behavior and performance is available for viewing and querying by anybody on the team that is concerned with quality. Visibility into product quality is radically better across the team as compared to common practices.

There is transparency across the team into the work of the developers and the QA role

Queryable Quality surfaces and shares detailed data to augment and help meetings and discussions.

2.9.7 Examples

Intranet portals are a common pattern at companies. Intranet portals into product quality information are a common pattern at software companies.

For the BankingAds app, the Queryable Quality pattern is implemented on an intranet portal that gives access to every team member concerned with quality of the SUT. Developers use it to research failures, e. g., when did the failures happen, how often, is there an emergent pattern or is there a correlation with other failures, etc. People in the QA role use it to watch app health as well as the health of their quality systems. Accountants doing work for Sarbanes-Oxley (i.e., company valuation, including software assets, and specific to the United States) have access to highly detailed, structured, direct, and highly credible information on the quality of the SUT and quality trends in the SUT.

If the BankingAds project team crosses geographies or cultures, Queryable Quality creates a new level of transparency in quality across the teams and vastly improves communication on quality issues.

With telemetry on the app, there is customer usage data which can supplement check results. The team can study correlations between changes in app behavior or performance with the telemetry data, through the Queryable Quality site.

2.10 Future Patterns for MetaAutomation

MetaAutomation will have more patterns in future. Just like the ones described here, the future patterns will address quality automation: the problem space between driving the SUT for verifications and measurements of functional correctness and performance, and the business that consumes that information. There will be patterns to address needs not covered by the patterns described in this paper.

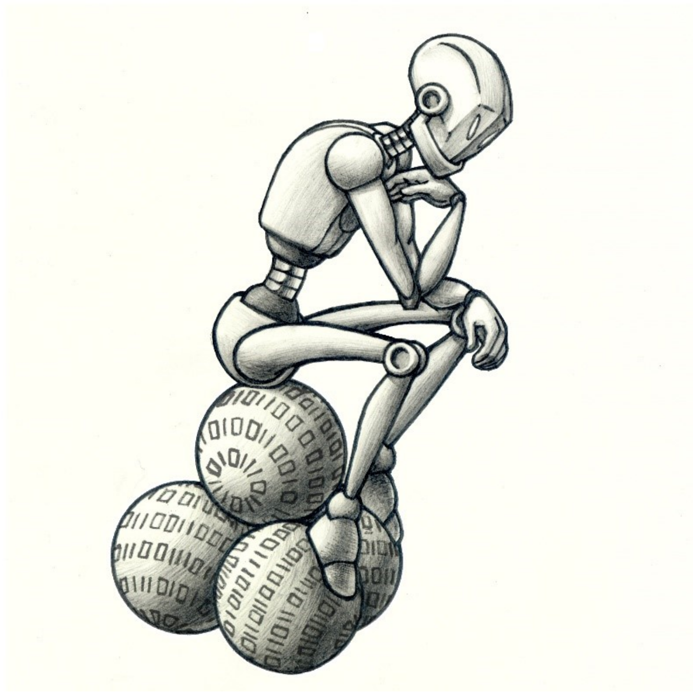


Fig. 25. Thinking about innovative solutions to add value in the quality automation space.

To become a living pattern language, MetaAutomation needs community; it needs people to not just educate on, execute and use the patterns, but also to confirm the existing patterns, evolve them or create better expressions of the patterns, and develop new ones.

Evolution and improvement in the existing patterns, and discovery, writing and teaching of new ones, will make MetaAutomation better and increase value to software quality practitioners and their teams.

For example, the Hierarchical Steps pattern could be a valuable tool in improving transparency into results provided by machine learning software.

For the reader who would like to get involved, or has questions on this or related material, please see the LinkedIn Group "Quality Automation" at https://www.linkedin.com/groups/13563800.

3. CONCLUSION AND FUTURE WORK

MetaAutomation gives an optimal and clearly-defined approach to measuring and communicating quality of deterministic or partially deterministic systems, and the deterministic foundations of probabilistic systems, from the positive perspective of "Does the SUT fulfill the functional requirements?" allowing software teams to develop software faster and at lower quality risk.

MetaAutomation is extensible. Patterns added in future will describe novel solutions in the quality automation problem space.

4. ACKNOWLEDGEMENTS

Thanks to my PLoP 2017 paper shepherd Neil Harrison, and members of the "Not Separateness" group at PLoP 2017:

- Lisa Hvatum
- Rebecca Wirfs-Brock
- Robert Biddle
- Tayyaba Nafees
- Tim Yao

Special thanks to my VikingPLoP 2017 paper shepherd Christopher Preschern for many points of excellent and timely feedback.

Thanks to my fellow VikingPLoP 2017 “Around the World” team members for excellent feedback at the conference:

- Malte Brunnlieb
- Veli-Pekka Eloranta
- Takashi Iba
- Klaus Marquardt
- Ville Reijonen
- Andreas Rüping
- Michael Weiss
- Joe Yoder

Thanks to Adrian Bourne for providing beautiful art work to visually express the pattern language and the patterns.

Special thanks to my shepherd for the 2017 PLoP conference, Neil Harrison, for his excellent and insightful feedback.

REFERENCES

Caspar, Mirko, Lippmann, Mirko, Hardt, Wolfram, Proceedings of the conference on Design, Automation & Test in Europe, March 2014

Cessna. 1984. 1967 Model 172 And Skyhawk Owner’s Manual, Version copyright 1984

Collins, Eliane Figueiredo, de Lucena, Jr., Vicente Ferreira, Software test automation practices in agile development environment: an industry experience report, Proceedings of the 7th International Workshop on Automation of Software Test, p. 62, June 2012

Goucher, Adam. 2009. Quality through Innovation. Retrieved June 1st, 2017 from http://adam.goucher.ca/?cat=3

Davies, Steven, Roper, Marc, What’s in a bug report?, Proceedings of the 8th ACM/IEEE International Symposium on Empirical Software Engineering and Measurement,p. 2, September 2014.

Elbaum, Sebastian, Rothermel, Gregg, Penix, John, Techniques for improving regression testing in continuous integration development environments, Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering, p. 235, November 2014

Garg, Deepak, Datta, Amitava, Parallel execution of prioritized test cases for regression testing of web applications, Proceedings of the Thirty-Sixth Australasian Computer Science Conference - Volume 135, February 2013

Harrison, Neil B.. 1999. The Language of Shepherding. Hillside Group. http://hillside.net/documents/language-of-shepherding.pdf

Herzig, Kim, Nagappan, Nachiappan, Empirically detecting false test alarms using association rules, Proceedings of the 37th International Conference on Software Engineering – Volume 2, p. 42, May 2015.

Jiang, He, Li, Xiaochen, Zijiang Yang, Jifeng Xuan, What causes me test alarm?: automatic cause analysis for test alarms in system and integration testing, Proceedings of the 39th International Conference on Software Engineering, p. 712-714, May 2017.

Kappler, Sebastian, Finding and breaking test dependencies to speed up test execution, Proceedings of the 2016 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering, p. 1136, November 2016

Leszak, Marek, Perry, Dewayne E., Stoll, Dieter, A case study in root cause defect analysis, Proceedings of the 22nd international conference on Software engineering, June 2000

Meszaros, Gerard. 1997. A Pattern Language for Pattern Writing. Hillside Group. Retrieved November 1st, 2017 from http://hillside.net/index.php/a-pattern-language-for-pattern-writing

Meszaros, Gerard. 2007. xUnit Test Patterns. Addison-Wesley

Myers, Myers. 1979. The Art of Software Testing. John Wiley and Sons, p. 5-8

Parveen, Tauhida, Tilley, Scott, Gonzalez, George, ACM-SE 45 Proceedings of the 45th annual southeast regional conference, p. 86, March 2007

Roseberry, Wayne. 2017. personal communication

Sebillotte, Suzanne. 1988. Hierarchical planning as method for task analysis: the example of office task analysis. Behaviour & Information Technology, 7:3, 275-293, DOI:10.1080/01449298808901878, 1988