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 for a team or company that is developing software. The technology side of this problem space includes automated operations on and measurements of software under development or maintenance for quality purposes. The business side includes customers of the quality information and other automated processes that depend on quality, for example, operations. MetaAutomation addresses the entire problem space between these two limits.

The focus of MetaAutomation is on verifying and communicating quality, i.e., answering the question for the business "Does the system meet functional and performance requirements?" for deterministic software and the deterministic foundations of probabilistic systems. Each of the patterns is based at least in part on existing patterns of human behavior, however the pattern language presented here is a novel combination of the patterns in a way that amplifies their value to the software business, relative to conventional practices of software quality.

MetaAutomation clarifies the business values of what an intentional, designed approach to measuring and reporting software quality with automation can achieve, as opposed to the conventional patterns of doing this which are rooted in misunderstandings and practices that do not scale to the ever-increasing complexity and importance of software.

Unit tests are out of scope because they are developer-facing and most often are not traceable to business requirements. MetaAutomation describes ideal automated end-to-end testing and bottom-up testing techniques.

The target audience is anybody doing, managing, or leading quality work with automated verifications and communication of functional and performance requirements.

The MetaAutomation pattern language has eight patterns: Hierarchical Steps, Atomic Check, Precondition Pool, Parallel Run, Smart Retry, Automated Triage, Queryable Quality and Extension Check.

General Terms: Software Quality, Automation, MetaAutomation, Test

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, and 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 aircraft, 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. For further information and clarification on some concepts and benefits covered briefly in this paper, reference to these samples (and, building, running, and changing them!) is encouraged. The samples need a free version of Microsoft Visual Studio, but most of the sample code, and the entire MetaAutomation concept, is platform-independent.

1.1 Why a Pattern Language?

The 8 patterns of MetaAutomation are more than a list or a catalog; they form a structure with defined dependencies, and together they define a whole, i.e., 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 expression of business value to motivate implementing the less dependent patterns.

Common misunderstandings and practices around software quality are preventing software from achieving levels of quality that are necessary today, and crucial tomorrow. MetaAutomation solves those problems. Current, pervasive,

and very costly misunderstandings include the persistent but obsolete meme that "The point of test is to find bugs," or that one should approach automated verifications just as one does manual testing [Myers 1979]. Traditional yet limiting practices include using the Linear Logging antipattern, i.e., linearly occurring log statements with inherently weak contextual information, with procedures of automated verifications where context is actually very important; logs work great for isolated events, but they are very poor for conveying context of a step in a procedure.

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 approach to quality automation rather than the historical limitations of the traditional approach.

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 began 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 value for the software business?

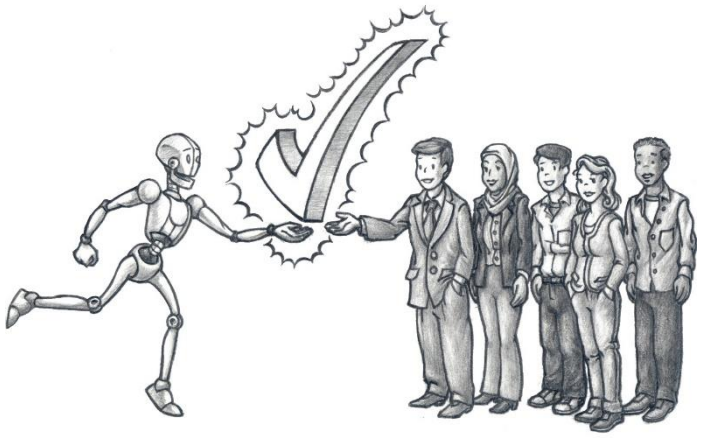

Fig. 1. This illustration shows the ideal of automation (represented by the robot) delivering software to the business (represented as the people of the business) that is correct for measured functionality, reliability, and performance.

The patterns of MetaAutomation are discoverable through existing practices, but the pattern language is designed with the software business in mind.

1.3 Overview

The pattern language map (Figure 2) shows the 8 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 interfaces with the business and the software under development.

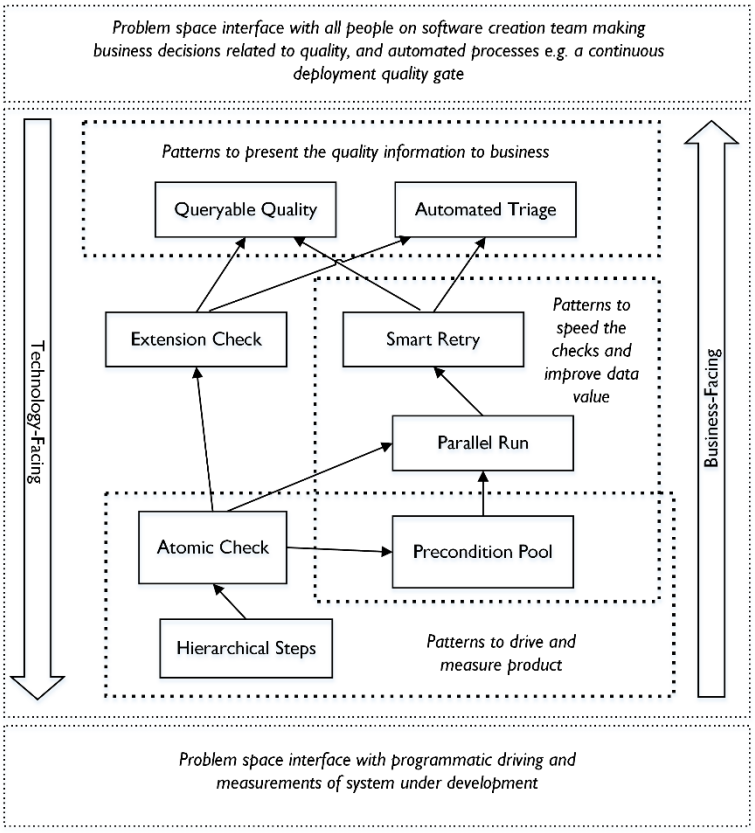

Fig. 2. MetaAutomation Pattern Language Map.

At the bottom of Figure 2 is the technology-facing context, i.e., 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 2 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 delivery.

Figure 3 shows graphically the dependencies of the patterns on each other. These relationships are the same as the ones described with prose in the pattern descriptions.

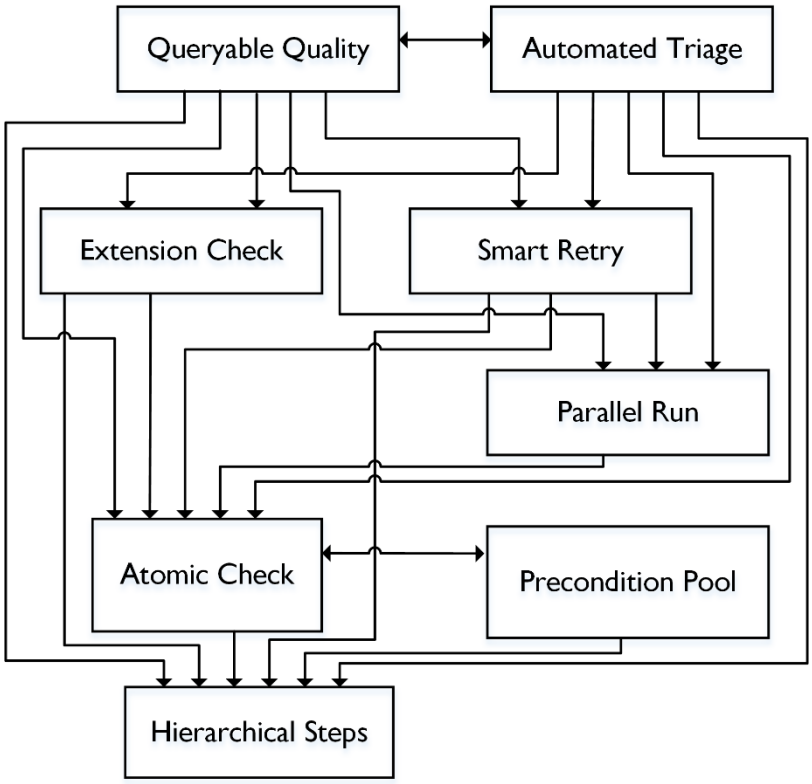

Fig. 3. Pattern Dependencies

The patterns to begin with are Hierarchical Steps and Atomic Check.

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 (or verify that the issue is fixed) 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 that may be unknown to the practitioner. For example, the Chained Tests pattern identified 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 2007a].

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 generated and recorded in correlation with executing a bounded and repeatable automated check.

1.4.4 Atomic Check

An Atomic Check uses as few steps as possible in driving and configuring the SUT, with all available dependencies in place, while verifying the linked functional requirement.

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. 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 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."

1.4.7 Business Requirement

A business requirement is something the product must do (or must *not* do), defined in an implementation-independent way and from a customer or business perspective. Business requirements link to functional requirements (see sections 1.4.6 above and 1.4.10 below).

1.4.8 Check

Check is a type of test where verification is limited to what is explicitly coded or implicitly verified as part of the code that executes the test. This excludes, for example, tests driven by people, because people see properties of the software product that are not explicitly targeted for verification or even necessarily known in advance what kind of issue they are looking for.

1.4.9 Counterforce

Counterforce is the opposite of a force in the patterns world, so is a potential reason to not apply the pattern.

1.4.10 Functional Requirement

A functional requirement is some required, measurable behavior of the SUT (see below) that is verified during testing, either manual or with automation. Functional requirements connect to Business requirements (see section 1.4.6 above).

1.4.11 GUI

An acronym for Graphical User Interface.

1.4.12 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 value of the quality data
- making both directed (i.e., push) and queryable (i.e., pull) communications of that data to the software business

The customers of quality automation include both people doing the software business 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?

1.4.13 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.

1.4.14 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.

1.4.15 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 the SUT, i.e., cascading failures are unlikely to cloud quality results.

2. THE PATTERNS OF METAAUTOMATION

Following are definitions for the 8 patterns of MetaAutomation.

2.1 Hierarchical Steps

2.1.1 Description

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

Fig. 4. Hierarchical Steps for a repeatable procedure.

In Figure 4, the robot is beginning the ordered tree hierarchy of a four-day hiking trip: the complete route on the map is the root node of the hierarchy, the distances ending with a tent or house form the child nodes at the 2nd level, and if the boulders are rest stops, they form 3rd-level or grandchildren nodes, with individual robot steps as children of those nodes at the 4th level of the hierarchy. The context for each node is structurally inherent. For example, the rest stop boulders marked on the map near the robot's head have their context indicated by their parent node: the 2nd day of hiking. The context for the 2nd day of hiking is marked by its parent node: the entire trip, with the house as destination.

2.1.2 Context

This pattern applies whenever persistence, communication, and/or analysis of a repeatable procedure is important, and enables detailed error handling and helps maintain the value of existing data derived from the procedure with occasional changes to the procedure. People use this pattern naturally when they do or think about performing tasks that are not extremely trivial.

2.1.3 Problem

People often record and communicate simple repeatable procedures with a linear list of steps. However, this is not adequate for complex or technical procedures or ones that need a high degree of transparency, because a linear list of steps does not enable adding more detail (e.g., if we needed to know more about step B or wanted to add data to that step), or error handling that shows context of a failed step, or anything more than the simplest context (e.g., step B happens after step A).

The logging idiom common to programming is effective for isolated events where the only context is a timestamp, but not adequate for details reported from a multi-step repeatable procedure. A linear list of steps, e.g., as one would get from output of a series of log statements, is not sufficient for the business needs of quality automation because most of the context for each step is lost; the resulting data is poor for analysis.

For the software business, valuable information on a repeatable procedure includes extensible detail, context, error handling and procedure modifications. All this must be preserved and communicated for review, query, analysis, error handling, and working with slight changes to the procedure.

2.1.4 Forces

Forces include necessary aspects of preserving and communicating business value:

- The procedure must be recorded, communicated, and queried in an extensible level of detail
- The procedure presents a high-level view and allows view drill-down to successive levels of detail
- For query, analysis and error handling, scope and context are clear for each step

2.1.5 Solution

Record the procedure in an ordered tree hierarchy of steps, rather than a simple linear list of steps. At any step in the hierarchy, a parent step is the context for child steps. Errors propagate up through parent steps to the root step, so an error is reported at every level represented in the hierarchy.

The sample implementations linked on http://metaautomation.net demonstrate and make this easy to do in C#, generating valid XML to express hierarchical steps, as represented in Figure 4 above.

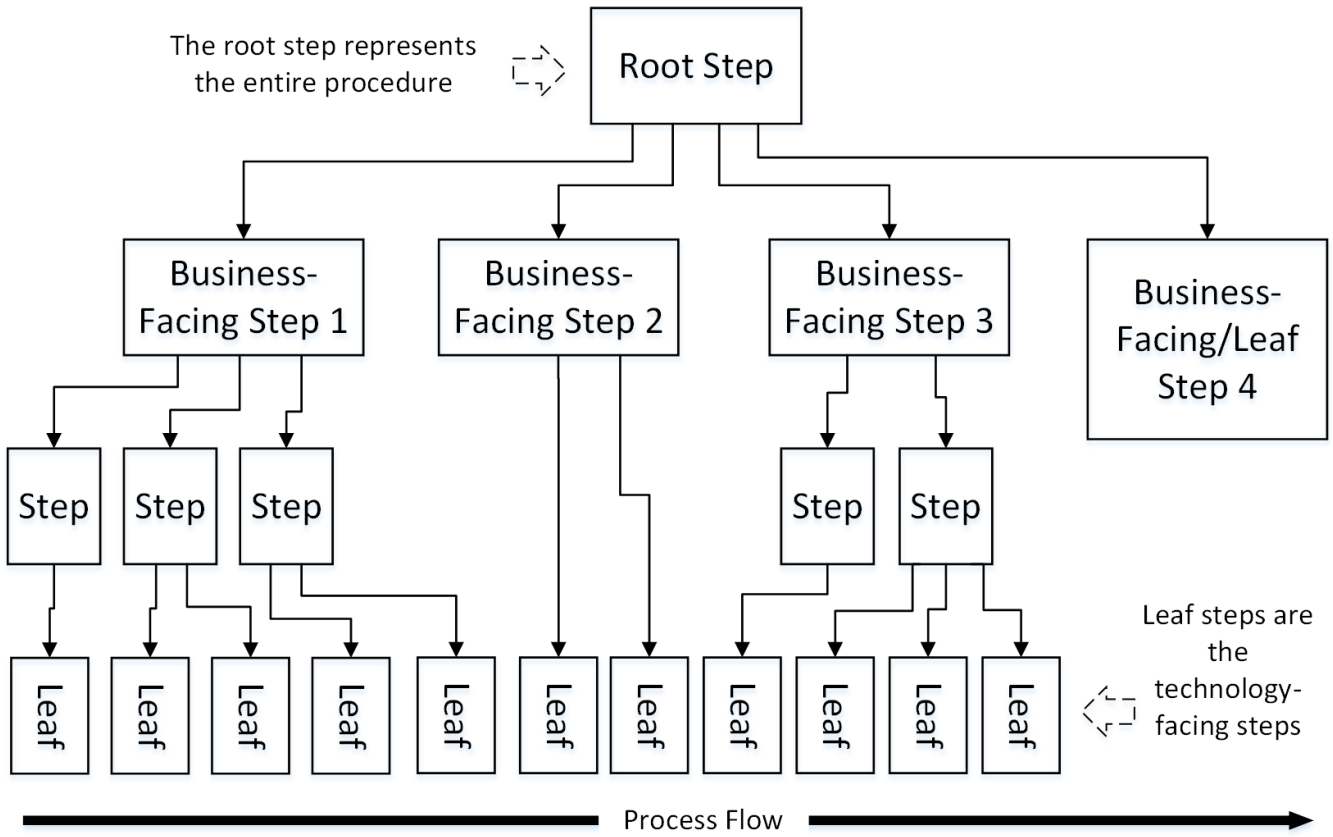

Fig. 5. shows the hierarchy of steps for a procedure

The technology-facing steps near the bottom of Figure 5 are the atomic steps of the procedure, i.e., the smallest steps that can fail.

2.1.6 Consequences

Applying the Hierarchical Steps pattern has these benefits, relative to the more traditional linear list of steps:

- The hierarchy describes the procedure with an expressive data structure that maps to how people think about and communicate the procedures, more closely than a linear list of steps does.
- If the procedure experiences a failure, the error propagation up through parent nodes to the root shows the step that failed at any level of abstraction represented in the levels of the hierarchy.
- Changes near leaf steps, or added details as leaf steps, do not effect most of the proceeding or following steps in the hierarchy in any way. The procedure description is therefore much more stable to changes or added steps, and previously gathered data on the procedure's correctness or performance therefore retains business value.
- Each node encapsulates the execution time for that node, so gives a clear context, measurement and storage of that important information. This both simplifies and improves data for performance metrics on the SUT.

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 described how steps of achieving a task can be optimally expressed in a hierarchy, as seen in this graphic from her paper [Sebillotte 1988] (Figure 6):

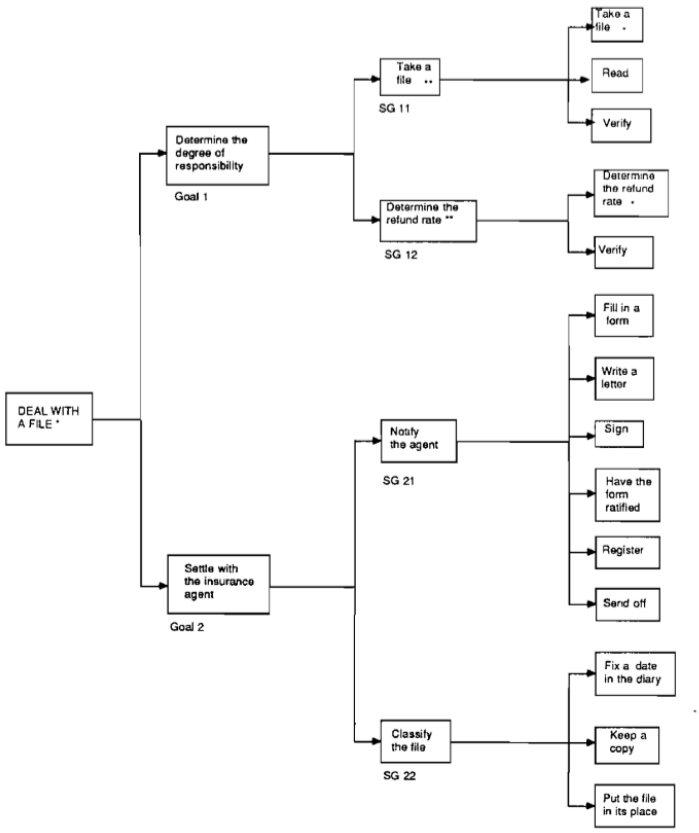

Fig. 6. Sebillotte's task hierarchy.

Figure 7 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 representing a higher level in the hierarchy and the lettered steps at each station representing the lower level, i.e., child steps of the station steps [Cessna 1984].

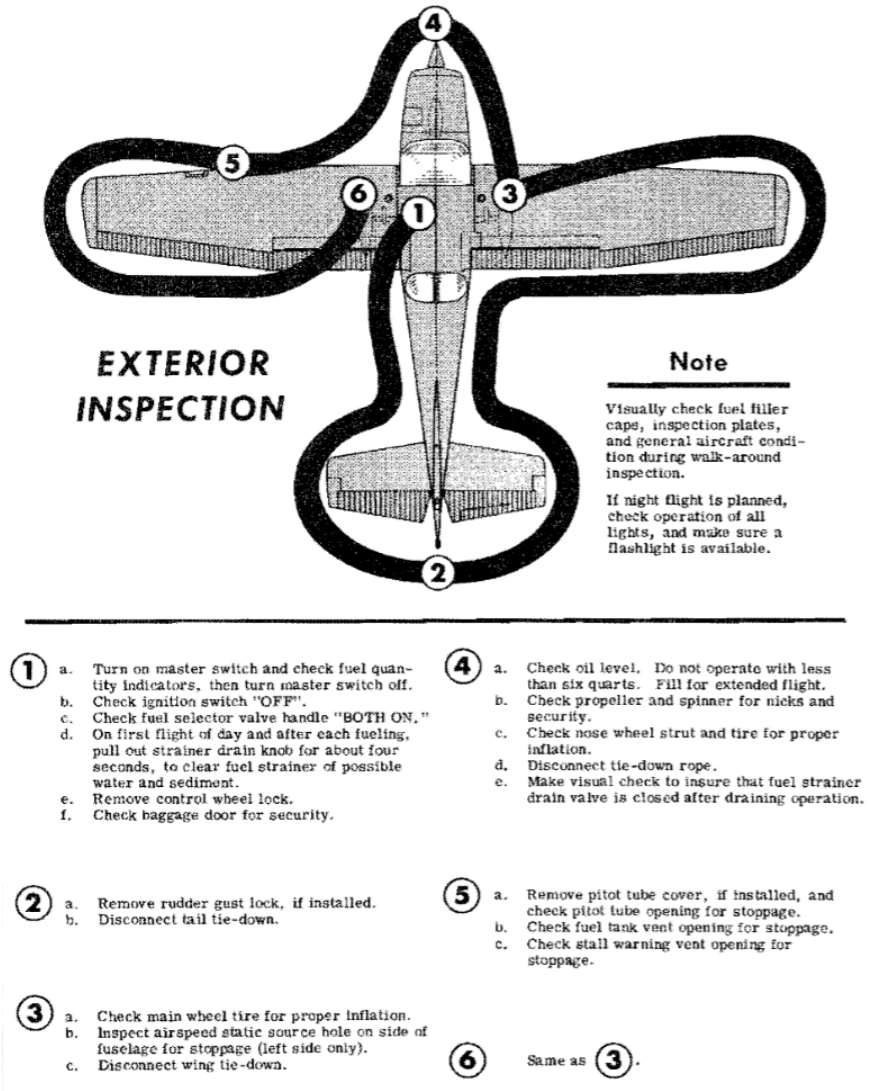

Fig. 7. Hierarchical Steps of preflight.

In the domain of running software, look to an example of buying a plane ticket online, which arranges the steps by web pages and controls on the pages. The itinerary request occurs on one page, where origin and destination are entered, one-way vs round trip, depart 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 greatly simplifies the ticket purchase process by communicating and leading the purchaser through a structure where the context for each step is clear; such clarity and ease-of-use would be impossible if the steps were encountered, for example, as a long linear list of isolated 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 the advertisements shown are selected based on (carefully anonymized) information from the end-user’s account balance, activities, and history. 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 maintains quality and helps the team develop the software faster. Applying the Hierarchical Steps pattern to automated verifications enables the team to record in full detail what the product is doing, including how many milliseconds each step took, and in case of failure, root cause of failure from the perspective of driving the product *and* what steps were blocked from measurement. This saves substantial amounts of debugging time, makes separate development of performance tests unnecessary, and communicates clearly 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 on http://metaautomation.net also show how hierarchical steps work with a very simple yet real-world software testing task.

2.1.8 Counterforces

This pattern applies to all repeatable procedures, but is usually left implicit for short lists of steps or verbal communication about some task.

For example, if I ask an assistant to gather some information for me, I might say: "Find Joe. Ask him for his best brief definition of a pattern, then report it back to me." This is superficially a list of three steps, and can be represented as such for casual communications. The implied hierarchy includes a parent step to the three step, that could be called "Get information from Joe." The first step "Find Joe" might include child steps of "Gather available information on Joe's current location," "Travel to Joe" and "Get Joe's attention." Such details are usually not expressed for very familiar or simple procedures, but they still exist implicitly.

Negative consequences of applying the pattern programmatically are the cost of keeping a hierarchical data structure, or using a programmatic system for doing so, although this cost is mitigated with complete open-source implementations described above in section 2.1.5.

2.1.9 What This Pattern Depends On

This pattern depends on a system for expressing an ordered tree, for example, the XML metalanguage.

2.1.10 What Depends on This Pattern

Most patterns of MetaAutomation depend on this pattern for trustworthy, detailed, analyzable data structures that describe software product behavior.

2.1.11 How This Pattern Relates to the Pattern Language

This pattern provides the data structure that enables all details of driving the SUT to be persisted and analyzed in an efficient and robust way, making in turn the Smart Retry, Automated Triage, and Queryable Quality patterns possible.

2.2 Atomic Check

2.2.1 Description

An instance of Atomic Check is a simple, focused automated procedure for verification of a functional requirement for the SUT. An atomic check can be end-to-end or bottom-up. The pattern requires that there be just one target verification or verification cluster, is as simple as possible (therefore indivisible and atomic) given the target, documents itself in detail (therefore needing the Hierarchical Steps pattern), and is independent of all other checks. The pattern also avoids creation and/or deferral of quality risk by including all existing dependencies be included, rather than stubbed or faked, as far as possible; please see Figure 10 and discussion, section 2.2.5, for an important reason why.

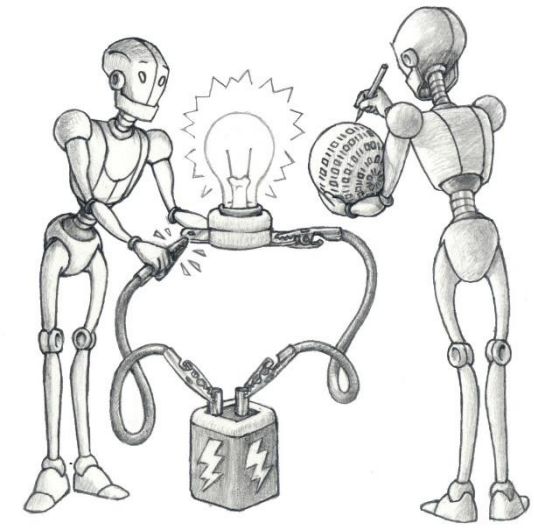

Fig. 8. Automation performing a check and recording results.

Figure 8 shows automated measuring and recording of an atom of functional product quality. The data recorded on the ball represented in the graphic is both detailed and structured.

2.2.2 Context

There is a need for the software creation team to write non-shipping code to drive the SUT, verify elements of software behavior, and report on this with detailed data on SUT behavior.

2.2.3 Problem

Common practices of automating procedures and measurements on the SUT often include checks that are unnecessarily complex, with many loosely-related measurements in a single check, and therefore relatively failure-prone and slow and/or with poor scaling characteristics. This problem becomes worse if checks are dependent on each other, as with the Chained Tests pattern [Meszaros 2007a]. For recording data, the checks tend to rely on log statements and other relatively ineffective methods.

As a result, quality data on the SUT is often lost through blocked measurements and, through poor recording, routinely unmeasured and lost. To recover the lost information, expensive manual debug sessions are needed. Any SUT data recorded from the product or generated by the test harness goes into the artifact of the check run, but the format tends to include long lists and/or mix data and presentation, and is therefore poorly suited to automated query or processing, further limiting the potential value of quality automation to the team.

Existing dependencies are often faked or even stubbed for speed and reliability, but this creates and defers quality risks when dependencies change or aren't fully modelled or tested, and potentially late-breaking high-risk changes to the SUT in response. MetaAutomation has approaches to speed (see the Precondition Pool pattern in section 2.3, and the Parallel Run pattern of section 2.4) and reliability (see the Smart Retry pattern of section 2.5) that reduce quality risk rather than create it.

2.2.4 Forces

For effective quality automation, the quality system must deliver results quickly and completely, and the results be as actionable as possible to both minimize re-work by people in the QA role and maximize trustworthiness in the quality system. Forces therefore include

- Up to a very large number of checks must run quickly and scalably

- Each step in each check must document itself, including pass/fail/blocked and milliseconds to completion, for every interaction with the SUT
- Each check must be traceable to requirements
- Risk must be managed by including Dependencies if possible

2.2.5 Solution

Design and implement checks as simply as possible given that each one must verify a single verification or verification cluster focused on a functional requirement, and with each one running independently of every other, with dependencies in place where possible.

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., the samples on http://metaautomation.net. This is represented with Figure 5 in section 2.1.5.

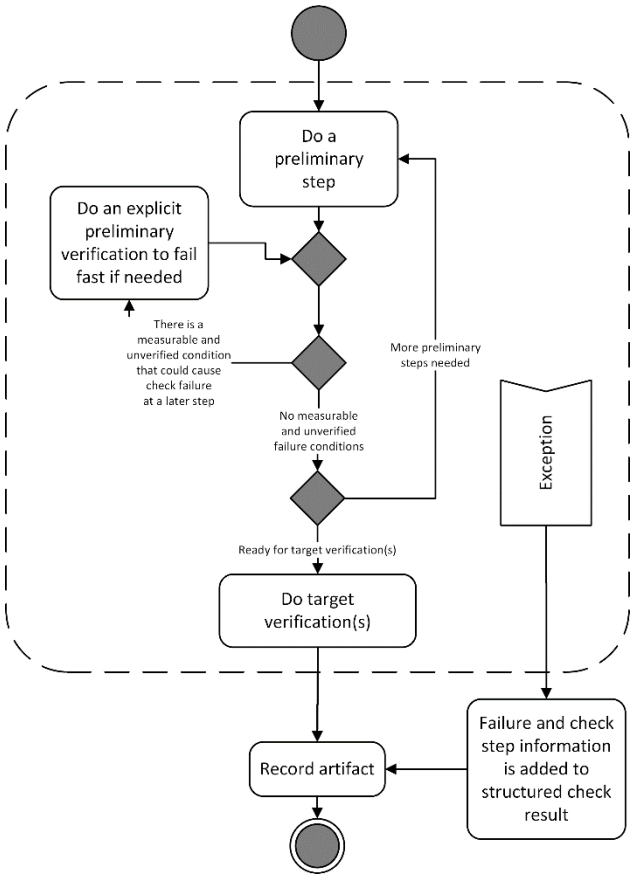

Fig. 9. The Atomic Check activity.

Figure 9 describes the Atomic Check solution.

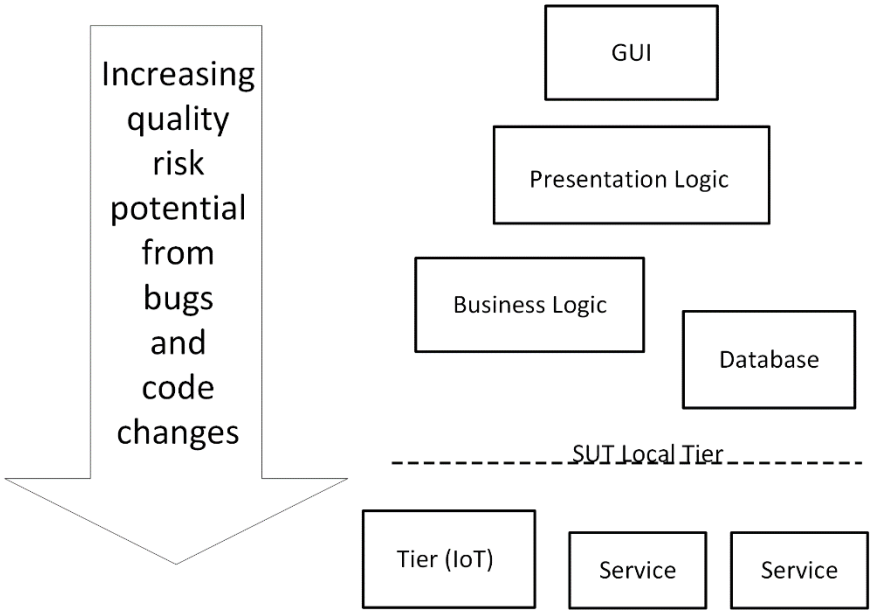

Fig. 10. The Importance of Dependencies

Figure 10 above shows the importance of including service and tier dependencies, even if they aren't part of the SUT; changes, error conditions in dependencies must be included in bottom-up or end-to-end testing to minimize quality risk. There is significant quality risk, too, if a faked dependency does not represent the behavior of the actual dependency exactly, so it's more effective to use the real one if available.

Issues in the most dependent layer – the GUI in the case of this simple example architecture – can be fixed with little quality risk. However, issues in the least dependent layers impact the entire system. For example, a slight change in a service can impact the entire SUT and potentially cause cascading bugs throughout the layers of the system.

For this reason, the Atomic Check pattern requires that all dependencies be present in the tested system, if possible, for end-to-end and bottom-up checks.

2.2.6 Consequences

The Atomic Check pattern helps the team by describing a standard and a procedure for defining an optimal check. This pattern is detailed, yet flexible and general enough that it can be applied to a wide variety of software products showing deterministic behavior and the deterministic foundations of probabilistic systems.

The business value of every implementation and every run of the Atomic Check pattern is traceable through the business and functional requirements for the product.

The simplicity of the checks makes them faster to run, more trustworthy and reliable, with check run artifact data that is structured and therefore more valuable, *and* at lower maintenance cost and risk as compared to conventional practices for software quality with automation.

Check runs create business value with every run, whether the check passes or fails, with information on what part of the check ran successfully and performance information on every step of the check that ran to completion. Since all of this data is in a hierarchy, it is very close to being "pure" data with no inherent presentation (i.e., there is no HTML or human language grammar); parsing, query and analysis of this data are fast, trustworthy and flexible.

On a check failure, the information on root cause of failure is complete from the point of view of the system driving the SUT, minimizing the need for reproducing the problem or cycling through a manual debug session before assigning an action item related to the failure, and maximizing the information used for indicating root cause or determining a course of action as a result of the failure.

On a check failure, if the procedure of the check has not changed and has run successfully to completion before, the artifact indicates the blocked steps of the check. This has value in showing where the quality risk may be, i.e., blocked SUT measurements related to the check failure.

2.2.7 Examples

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

Simple, focused checks are used by practitioners in the field, 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 2007b].

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.

The sample implementations on http://metaautomation.net also show how implementations of simple atomic checks.

2.2.8 Counterforces

This pattern is about automated verifications of software behaviors. Applying this pattern to manual software testing is not recommended.

This pattern can be adapted to fuzz testing, but only part adds value to model or state-based testing, or unit tests, or tests with existing dependencies stubbed out.

A negative consequence of applying this pattern is that, given that most checks target just one functional requirement and the Chained Tests antipattern is disallowed [Meszaros 2007a], the overall number of checks might be larger than otherwise. If the Parallel Run pattern is not implemented, the check run might therefore take more time, although the quality of the data from the check run would be much richer and more focused than the Chained Tests case.

2.2.9 What This Pattern Depends On

Atomic Check has a strong dependency on the Hierarchical steps pattern to be self-documenting in complete and manageable detail.

Atomic Check also has a weak dependency on Precondition Pool, in the sense that Precondition Pool helps any Atomic Check implementation be simpler, more focused, and more independent of the other checks.

2.2.10 What Depends on This Pattern

Smart Retry, Automated Triage and Queryable Quality all depend on short, independently-running checks. These three patterns and Extension Check depend on the completely self-documenting artifact from Atomic Check.

Parallel Run depends on Atomic Check because the checks must be as fast as possible and all independent of each other.

Precondition Pool depends on Atomic Check because, given that each check is independent of all the others, so the dependencies of the checks are simple and external to the check run itself.

2.2.11 How This Pattern Relates to the Pattern Language

This pattern describes the requirements for designing checks for the strongest business value for quality automation, and makes the six dependent patterns possible. From the point of view of the QA role, Atomic Check describes optimal check design.

2.3 Precondition Pool

2.3.1 Description

This pattern keeps pools of resources, i.e., preconditions for the checks to run, in a ready state so that they are immediately available to the checks during the check run. This simplifies the checks and focuses the data on the target verification. In addition, if the pooled resources already exist in sufficient quantity at the launch of a check run, overhead to run the pools will be minimal during the check run and the run will complete faster to deliver quality data sooner.

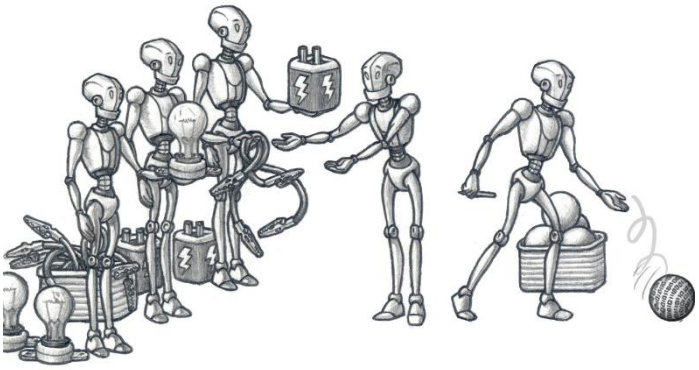

Fig. 11. Three Precondition Pool instances.

Figure 11 represents three such pools: one for the light bulbs, one for the wires, and one for the batteries.

2.3.2 Context

The checks have some dependency requirements, including an environment in which to run, and potentially things like documents of a known state, or user accounts of certain roles and/or configurations.

2.3.3 Problem

Checks may be slower, more complex, and with more potential points of failure than necessary if all resource setup is done in sequence for every check run.

2.3.4 Forces

The checks have some runtime dependency requirements, including an environment in which to run, and potentially things like documents of a known state, or user accounts of certain roles and/or configurations. They need to be as fast, simple and reliable as possible and focused on the target verification.

2.3.5 Solution

For each check, consider managing the needed check resources externally; as long as the resources concerned are not being measured for quality in the check, and they can be managed externally and asynchronously to the check, move them outside the check with Precondition Pool.

Each implementation of Precondition Pool manages one type, role, and/or configuration of external resource. For example, the common pattern of managing many computing environments in which checks are executed can be understood as such an implementation. If there are different types of environments needed for the check runs, e.g., with different installed or running resources, each different type can be managed in one Precondition Pool implementation; each such implementation can manage any number of environments of the appropriate type.

Any number of resources of the needed types can be queued up and available before the check run has even begun; this speeds the check run, and improves the scale with resources. Any errors related to managing the resources are also managed within the Precondition Pool implementation, and therefore have little or no impact on the checks.

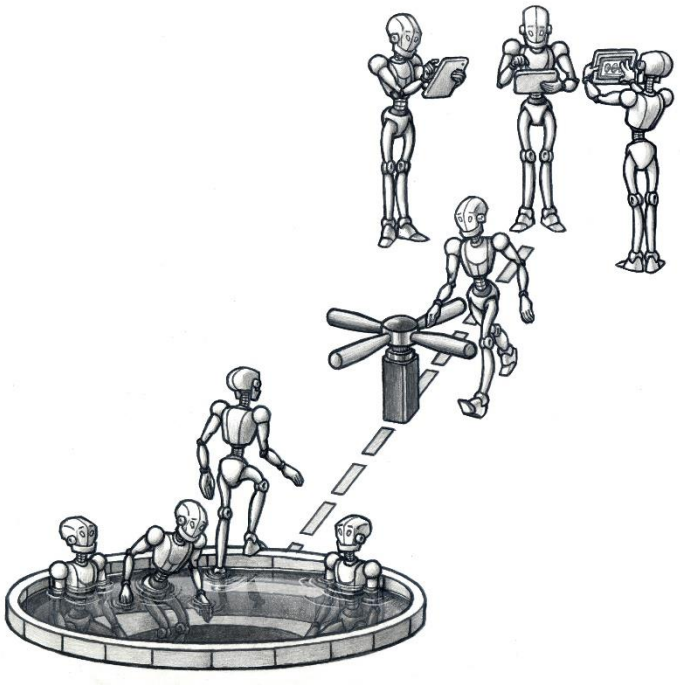

Fig. 12. Precondition Pool resource flow for an object type

In Figure 12 above, robots represent the resources that are checked out of the pool for help in running a check, then released back to the pool to restore state.

2.3.6 Consequences

Depending on how many external resources are identified and managed in Precondition Pool implementations, the checks will be simpler, faster, more trustworthy and with simpler artifacts that are more focused on specific parts of the SUT and less prone to failures in SUT dependencies. The artifact data will therefore be of higher quality and more valuable to the business.

2.3.7 Examples

The Setup and Teardown phases of the Four-Phase Test pattern identified by Meszaros are precursors to Precondition Pool because the 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. With Four-Phase Test, however, the phases may happen on one thread and dependency management happens in-line, which has negative consequences on check complexity, reliability, performance, scale, and the quality of the artifact data.

The Precondition Pool pattern is applied where checks are 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 various 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 gives the team opportunities, because now it's clear that (depending on the SUT and dependencies) there are probably other resources that can be moved out to a Precondition Pool instance to create faster, simpler and more effective checks.

Returning to the BankingAds app, Precondition Pool applies to managing the many environments in which the checks can run scalably, as well as customer accounts of various types and balances. Making customer accounts available to the checks, and restoring or rebuilding them as necessary, makes the checks simpler and faster and thereby shortens the overall check run and improves the quality of the data that is created by the check run.

2.3.8 Counterforces

Precondition Pool should not be applied if the costs outweigh the benefits, for example, if the number of checks or the computing resources available to run the checks is small. This might be true because the SUT is simpler and lower-impact than, e.g., the BankingAds example, or app quality is low priority because it is not important to any end-user.

Negative consequences of applying the pattern include

- The engineering overhead of implementing the pool(s)
- The overhead of managing the resources in case of failure is simply moved from the quality checks themselves to the Precondition Pool implementations, so another system is needed to manage those failures
- If demand for pooled resources during a check run overwhelms the resources that were prepared before the check run is launched, then computing resources overhead is simply moved from the check run to the Precondition Pools, and the acceleration of the check run that a PP enables will diminish during the check run

2.3.9 What This Pattern Depends On

Precondition Pool depends on Hierarchical Steps for detailed, useful artifacts from creating objects for the checks, that help in turn resolving any errors in creating the objects for a pool.

Precondition Pool depends on Atomic Check because the checks must be independent of each other if the dependencies of each check are to be simple, scalable and independent of the checks themselves.

2.3.10 What Depends on This Pattern

Atomic Check, Parallel Run, Smart Retry, Automated Triage, and Queryable Quality all depend on Precondition Pool because the latter can make checks faster and simpler and check runs more numerous. These five patterns plus Extension Check depend on the quality data that Precondition Pool helps make possible.

2.3.11 How This Pattern Relates to the Pattern Language

This pattern encapsulates a way to speed the checks further and improve data, improving the effectiveness of the quality automation.

2.4 Parallel Run

2.4.1 Description

Parallel Run is about running checks in parallel to use available computing resources. Since each atomic check runs independently of every other check, the checks can be run in parallel across different machines, virtual machines or processes, and thereby run to completion faster.

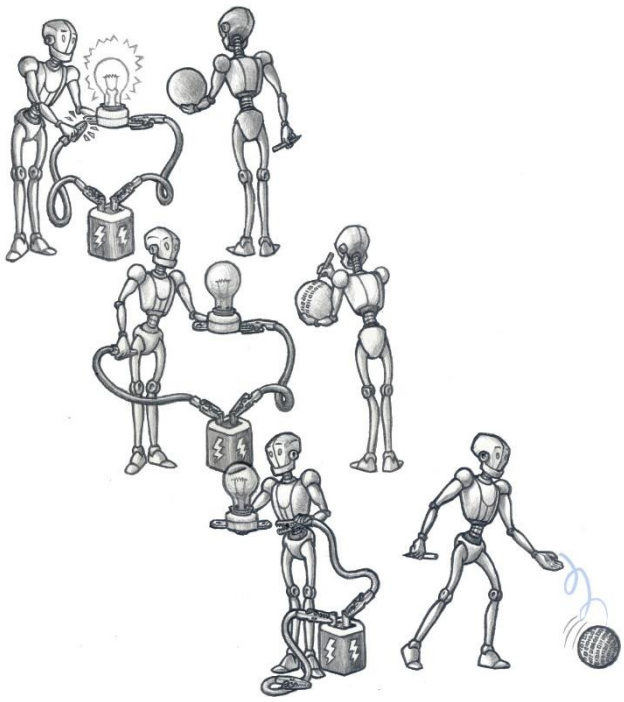

Fig. 13. Three instances of automation processes running in parallel.

Figure 13 represents three environments to run the checks in parallel.

2.4.2 Context

With easily-available computing power and checks that are independent of each other, there is an opportunity to run checks in parallel to speed the check run.

2.4.3 Problem

Checks must run quickly, in large numbers, especially when the team is using check-ins gated with end-to-end quality checks. Given increasing availability and declining cost of computing resources, the team must be able to use them to scale the check run.

2.4.4 Forces

Large numbers of checks must deliver results quickly, therefore they need to scale with resources.

2.4.5 Solution

The solution is to parallelize the check runs across different virtual machines or OS instances, in the same way that job management tools parallelize batch runs across computing resources.

Without parallelization, the checks all must run sequentially, as the Figure 14 shows:

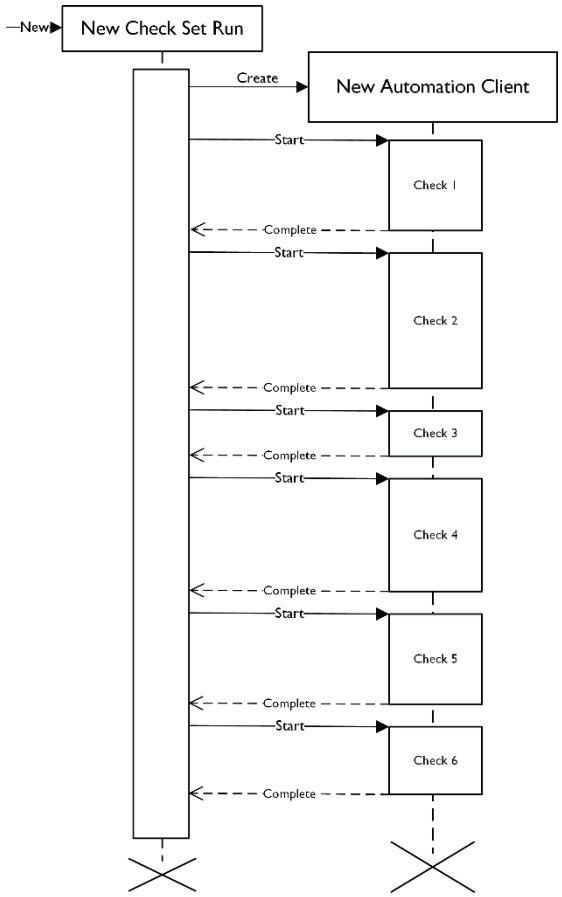

Fig. 14. Checks running sequentially.

With parallelization, the checks can run on an arbitrary number of clients, so they run faster; given a large number of such checks and a potentially large number of clients, the speed at which the checks run is almost arbitrarily fast. Figure 15 shows the speed increase with the same checks run across three clients:

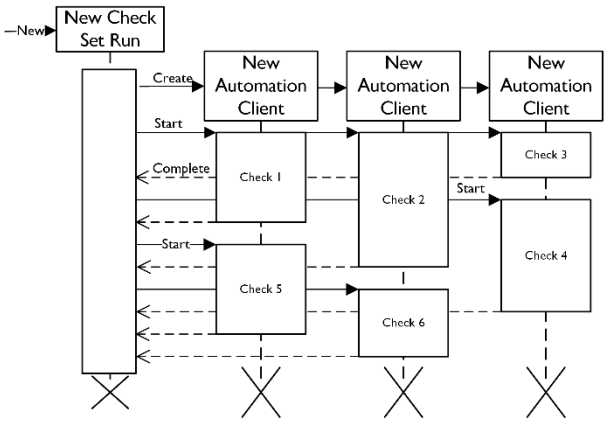

Fig. 15. Checks running in parallel.

2.4.6 Consequences

The check run can complete in an almost arbitrarily short time period.

Given a significant number of end-to-end and bottom-up checks and significant computing resources, and given that such checks tend to take significant time, the speed benefit of running the checks in parallel easily outweighs the small added overhead of managing the distribution of the check runs.

2.4.7 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 almost arbitrarily fast.

2.4.8 Counterforces

This pattern does not apply if the Chained Tests antipattern is used [Meszaros 2007a].

Negative consequences include the overhead of creating the shared resources, and preventing contention and race conditions through locking and job distribution.

2.4.9 What This Pattern Depends On

This pattern depends on these aspects of the Atomic Check pattern:

- The checks are all independent of each other at runtime.
- The checks are as fast and as simple as possible.

2.4.10 What Depends on This Pattern

Parallel Run runs many checks and delivers detailed, trustworthy quality data on time. Smart Retry, Automated Triage, and Queryable Quality depend on this pattern for copious and robust quality data, delivered quickly.

2.4.11 How This Pattern Relates to the Pattern Language

This pattern scales check runs with resources to generate more data faster on SUT quality, and in turn make the four dependent patterns more effective and valuable.

Parallel Run is the pattern that enables gated check-ins with bottom-up and end-to-end checks, thereby protecting developers from the risk of blocking each other with quality issues that the quality automation looks for, and ensuring that quality always goes forward.

2.5 Smart Retry

2.5.1 Description

The Hierarchical Steps and Atomic Check patterns ensure that every step of the check procedure, from the perspective of the quality automation driving the SUT, is recorded in pure data. In addition, the hierarchy of the data generated by the check supplies context-rich placeholders for any product instrumentation, stack traces and other exception information on failures, etc. Figure 5 of section 2.1.5 shows this. Short of reproducing the problem and debugging through the code of the SUT for additional information, as a dev might do, all the available information is already present and recorded in a pure-data hierarchical format, e.g., in a defined grammar of XML. In case of check failure, the root cause of failure when driving or measuring the SUT is caught by the "Exception" activity in Figure 9 in section 2.2.5 above.

Smart Retry enables real-time decisions on whether to retry a failed check based on e.g. this information:

1. On which application tier the check failed, if applicable
2. Which technology-facing step in the hierarchy is root cause of failure
3. Stack trace of a caught exception, if applicable
4. Root cause of failure as recorded in the artifact or artifacts of the one or two previous runs of the check
5. Whether a failure for specific root cause has been reproduced
6. How many total tries have occurred during the current check run for this check

Smart Retry improves the quality of artifact data from a check run and greatly reduces both blocked quality measurements and check failures that would, if they were not enhanced with a retry of the check, become a non-actionable check failure and therefore a costly distraction to the business.

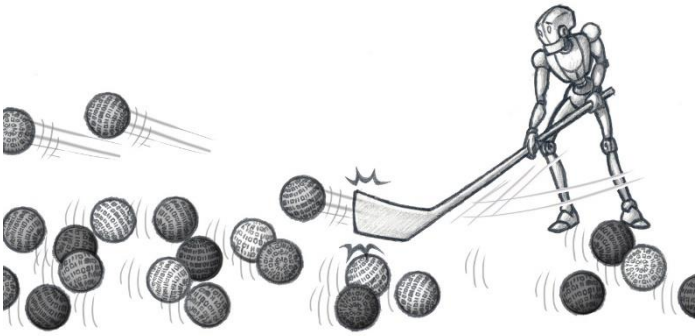

Fig. 16. Automation choosing checks to retry.

Figure 16 shows the Smart Retry implementation selecting checks for retries based on their results. All resulting artifacts from the checks are saved, whether or not they are retried; data from initial tries and re-tries of a given check are all packaged together to show what happened with the re-try (or retries) of a check.

2.5.2 Context

Sometimes checks fail, and some of those check failures are not actionable by the business because the failure might be intermittent and due to an external resource or a race condition in a GUI or web browser.

2.5.3 Problem

MetaAutomation focuses on end-to-end and bottom-up checks. (See section 2.2.5, Figure 10 and discussion, for an illustration of the importance of bottom-up checks.) Fake interface implementations or stubs are recommended only when the actual dependency is not available for some reason, in order to avoid deferral of quality risk. Check failures due to external dependencies may therefore be unavoidable, just as check failures due to race conditions in a GUI are sometimes unavoidable. These “false positive” events cause a notification that the check reports something actionable, but the notification is false; on manual examination, which is expensive to the business, it turns out that the check failure is not actionable.

Such events tend to unnecessarily de-prioritize individual checks or reduce trust on all check failures, because the ultimate course of action for the business may be to ignore such failures anyway. As the business works to minimize such costs, the flow of quality information – including real, actionable check failures – becomes less efficient.

2.5.4 Forces

The business can maximize productivity, communication, and trust by shielding people from being interrupted by non-actionable check failures, also known as “false positives.”

In addition, action taken on a check failure may depend on whether or not a given failure can be reproduced with the same root cause, so automated generation of this information is valuable.

2.5.5 Solution

Eliminate notifications related to non-actionable check failures.

Immediately upon failure of any check, use factors described in the Description section 2.5.1, above, to decide whether to retry the check. With a system that retries a failed check, automation decides at check runtime whether the

failure is a candidate for retry and whether the exact failure has been reproduced. If retry is decided and the check passes on retry, the artifact of all runs is persisted, including the failure, but to downstream quality automation the check appears to have passed. A flaky check that fails and then passes will not interrupt anybody's workflow; the flaky check problem is therefore solved.

Actionable check failures might include a product bug that was reproduced with the retry, or a reproduced timeout that recommends a timeout adjustment, or a check code issue.

Smart Retry bundles the artifacts of retried checks in a given check run together, so in case a notification Is needed, it improves the check artifact data that is used by the Automated Triage pattern to direct notifications. Whether or not a push notification takes place, the bundle is viewable and analyzable as such according to the Queryable Quality pattern.

Figure 17 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:

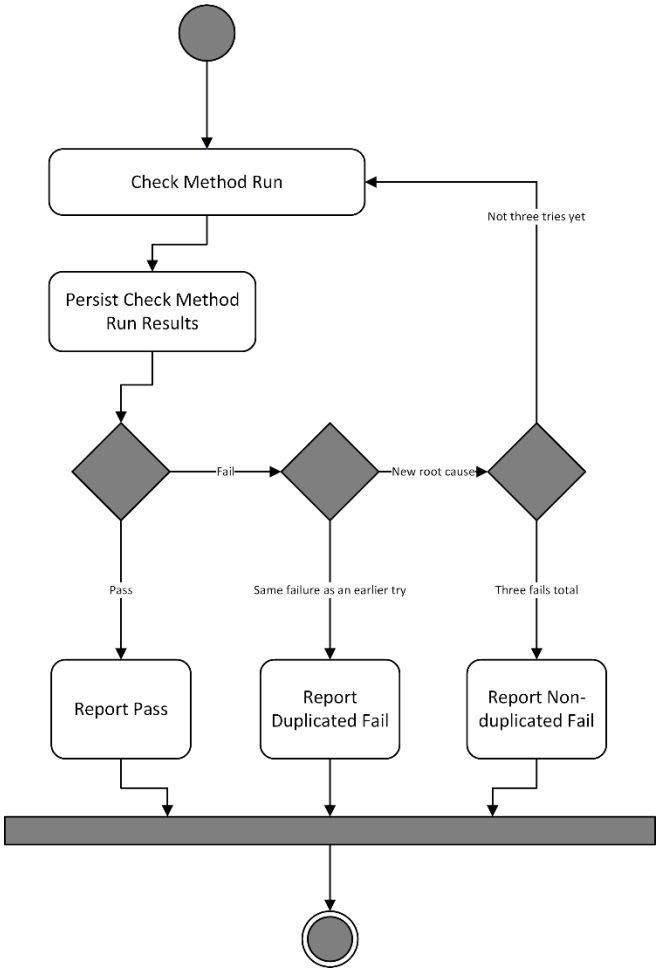

Fig. 17. The Smart Retry activity, in the case where all checks may, in case of failure, be candidates for retry.

A slightly more complex implementation and representation considers that for some failure cases and some products, retry is not desired and would be disabled by default based on the detailed artifact from a failed check. For example, a non-deterministic failure due to SUT code might be immediately actionable, and not a candidate for retry.

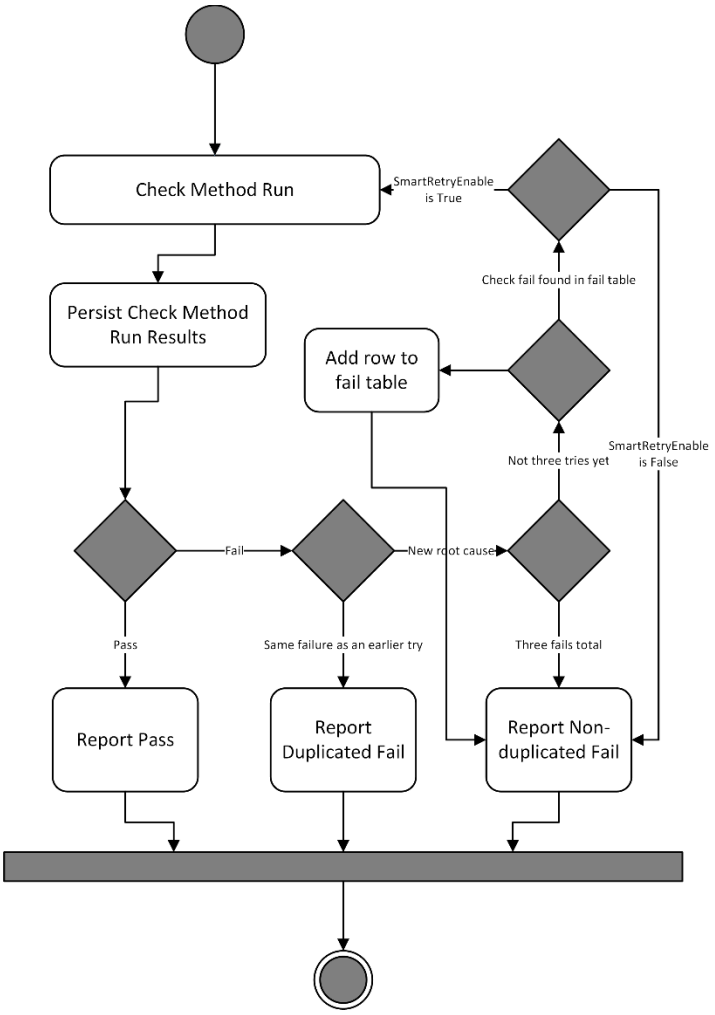

Fig. 18. The Smart Retry activity, in the case where smart retry is disabled by default

Figure 18 shows the case where smart retry is not desired for some or most aspects or behaviors of the SUT. A table of root cause information, with the manually enabled SmartRetryEnable flag enabling retry, determines whether or not the retry happens.

2.5.6 Consequences

With the Smart Retry pattern, we can finally solve the flaky test problem: intermittent check failures that are due to, e.g., race conditions in the SUT, are still reported as simple check failures and are therefore still important action items for the developers, but intermittent failures due to external race conditions or other failures do not get automatically reported at all if the first or second retry of the check succeeds. Therefore, although the artifact data from failures is all persisted for later analysis, the check run overall can succeed and nobody in the business has his or her workflow interrupted due to check failures that are not actionable.

The resulting value of and trust in the check failures that are reported as action items to the business (through the Automated Triage pattern, section 2.6, and the Queryable Quality pattern, section 2.7) is greatly increased.

Failures that are unique or appear to be non-deterministic and show root cause outside of team ownership are less of a problem for the team with Smart Retry because, with effective quality automation, they will not appear in anybody's workflow with exaggerated priority. Nobody's flow needs to be interrupted. However, the data resulting from the check run that failed with a one-off failure is all present and available for later analysis; see the Queryable Quality pattern, below.

Smart Retry helps the business run more efficiently by automatically collecting, through data-driven check retries, more data that informs whether a failure is immediately actionable and, if it is actionable, what that action might be.

2.5.7 Examples

Microsoft, Google and others use a pattern called "Retry" which is to simply retry a check, up to three tries total, on failure. 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. However, the risk of applying this pattern is that 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; of course, the data to make such a distinction is probably not available to the automation anyway because there is no adequate system to persist this information (as the Hierarchical Steps pattern addresses above in section 2.1). The data lost in this way can hide real failures in the SUT.

In the Office team at Microsoft, the complexity of the SUT is such that it includes nondeterministic business logic conditions (please see the Extension Check pattern of section 2.8), so the Retry pattern is used here as well [Roseberry 2017a].

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

For the BankingAds example, given that the end-user interface for the app is a web browser, there will likely be failures due to race conditions in the client interface. These are candidates for retries according to Smart Retry. If the timeout failure is reproduced at the same 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 immediately actionable.

Smart Retry uses the detailed information provided by implementing the Hierarchical Steps pattern to discriminate between check failures that should be retried, and checks that should not be retried, and will determine quickly whether a failure has been reproduced at the same leaf step of the check.

2.5.8 Counterforces

Smart Retry should not be applied to systems such as avionics where every failure may be immediately actionable, and might not be applied (or, simply disabled) on highly deterministic business-logic layers of the application, depending on the app and the types of failures seen.

Negative consequences include the overhead of implementing it, and the very small runtime overhead of grouping, linking and wrapping check results for a retried check.

Also, although Smart Retry never prevents results from being persisted, if configured incorrectly it might have the effect of *temporarily* hiding actionable issues of product quality.

2.5.9 What This Pattern Depends On

Smart Retry depends on Hierarchical Steps for trustworthy, detailed and specific data on point of failure of a check from the point of view of the system driving the SUT through the check.

Smart Retry depends on Atomic Check for fast, simple, independent checks with few points of failure.

Smart Retry depends on Parallel Run for checks that run in parallel, so that a retry can happen with minimal impact on the overall check run.

2.5.10 What Depends on This Pattern

Automated Triage and Queryable Quality depend on Smart Retry to reproduce persistent failures and hide non-actionable transient ones to improve trustworthiness and value of actionable data.

2.5.11 How This Pattern Relates to the Pattern Language

This pattern is about improving data quality and impact facing the software creation business.

Smart Retry solves the "false positive" problem of automated checks.

2.6 Automated Triage

2.6.1 Description

Automated triage uses artifact data from an actionable check failure (or collected failures of the same check; see the Smart Retry pattern above in section 2.5) to decide who needs to be informed and with what data, and then dispatches communications to that user or distribution list.

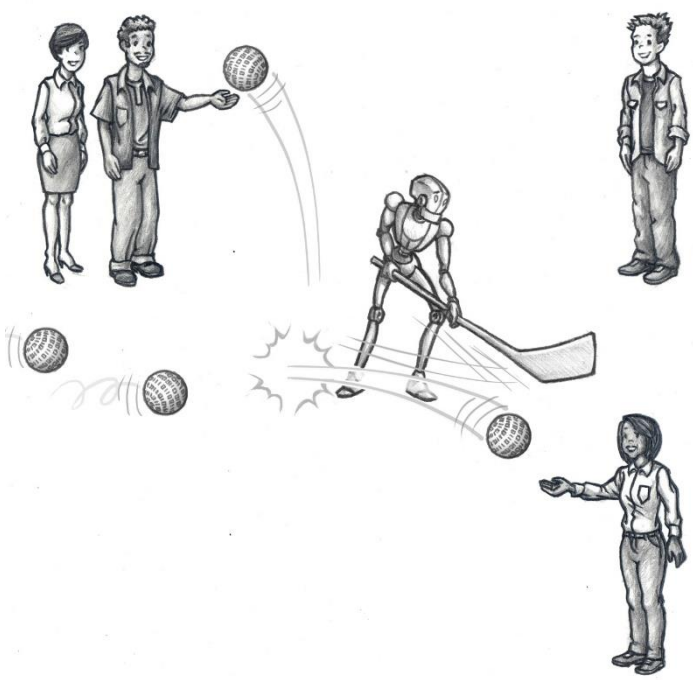

Fig. 19. Automation directing notifications.

Figure 19 shows the check artifacts dispatched by an Automated Triage implementation to different stakeholders. With detailed artifacts that show exactly what happened with the SUT, a target for the action item is easily identified.

2.6.2 Context

Sometimes people and processes in the business must be notified immediately of quality issues, but too many such notifications or ones directed to the wrong people or distribution lists tend to be ignored, or take valuable business time determining whether the implied action item is for the recipient.

2.6.3 Problem

In a QA team or group of people who have responsibility for automation that measures the SUT or an immediate responsibility for the results of those measurements, a common pattern is automated emails related to check results, with an emphasis on when the checks fail. Unfortunately, these automated emails notify everybody on the distribution list of every such notification event; there is no discrimination based on whether something is actionable or who might be the owner for an action item. As a result, people tend to ignore these emails, at least for a time, to defer having their work days interrupted by what is mostly likely not an action item for them anyway.

The result is inefficient use of business resources, and inefficient flow of important quality information on the SUT.

A manual triage may help, but tends to be repetitive and to wait on a meeting of people for triage or the same offline triage of a responsible person doing it at his/her workstation, introducing delay which may itself be expensive as responsible engineers are more likely to have moved on to other tasks (which brings, in turn, annoying and expensive randomization to the team). This dependency on people may be worse for teams distributed geographically over different time zones, because people typically work during certain hours of the day but not others.

2.6.4 Forces

The quality system must avoid unnecessary notifications, and target the correct ones appropriately. For notification recipients, this will improve trust in and responsiveness to the quality notifications that they do receive.

2.6.5 Solution

The answer is to develop or use a rules engine to choose recipients or recipient lists by comparing artifact data and Automated Triage configuration data, and then send notifications to only those recipients.

The data used as input to such a rules engine for a given notification includes similar information as that described in section 2.5.1 above for the Smart Retry pattern, in addition to any retry information for a retried check. Since the artifact of a check run is pure data, e.g., valid XML, by the Hierarchical Steps pattern, automated analysis is efficient, transparent, and reliable.

2.6.6 Consequences

Directed notifications are sent only to people for whom the notification is actionable.

This drastically reduces the number of automated notifications that are sent out, and greatly increases the average importance and suitability of automated communications received by team members on quality issues. Therefore, team members are much more likely to pay attention when such a notification arrives at their work station, needed actions are not delayed, and the business runs more efficiently.

The notifications also include links to the check data on the intranet, or queries of this same data. Please see the Queryable Quality pattern, section 2.7.

2.6.7 Examples

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

For the BankingAds app, Automated Triage directs communications to people in the QA role, for example in the case of a web page object timeout that was reproduced by the Smart Retry pattern, or to developers, for example in case of an incorrect bank balance.

2.6.8 Counterforces

Automated Triage is disabled for private builds, or configured to send notifications to the developer owning that private build.

Automated Triage is of limited utility to a very small software team, or one that discourages automated communications.

Negative consequences include the overhead of implementing the pattern and maintaining the rules that determine notification targets based on failure root cause and, in case of retry, additional data around retries of a failed check.

2.6.9 What This Pattern Depends On

Automated Triage needs Smart Retry to reproduce errors in the case of reproducible errors, or not in the case of a flaky test, and therefore increase the quality of the data and confidence in the value of the data from notification recipients.

This pattern also needs the detailed and trustworthy step data from Hierarchical Steps.

This pattern depends on Atomic Check for the high-quality data from checks that are as simple as possible and independent of each other.

For the pattern to deliver well for the team, it needs many checks and check runs, a benefit of applying Atomic Check, Parallel Run, and Precondition Pool as well.

For systems where the Extension Check pattern is used, Automated Triage depends on the data of a failed Extension Check implementation to send notifications.

Also, see section 2.7.10 for the Automated Triage dependency on Queryable Quality.

2.6.10 What Depends on This Pattern

The direct beneficiaries and dependents on this pattern are people and processes in the software business who receive the targeted notifications of Automated Triage.

The Queryable Quality pattern depends on the notifications of Automated Triage to provide easy and germane points of entry, through intranet links, to the Queryable Quality portal.

2.6.11 How This Pattern Relates to the Pattern Language

This pattern pushes action items from quality automation to the business.

2.7 Queryable Quality

2.7.1 Description

Queryable Quality is about presenting the quality data to the business for viewing, query, and analysis. Business owners, accountants, executives, developers, members of the QA team or anybody concerned with quality of the SUT can view and query the data. For a given check (or, a given run of a given check) the data hierarchy is represented broadly in Figure 5 (section 2.1.5). In case of a failed check, the initial failure may be grouped with a subsequent success or other failure(s) as described by Figures 17 and 18 in section 2.5.5.

An implementation could be an intranet web site for the business with role-based security for users who need quality information on the SUT.

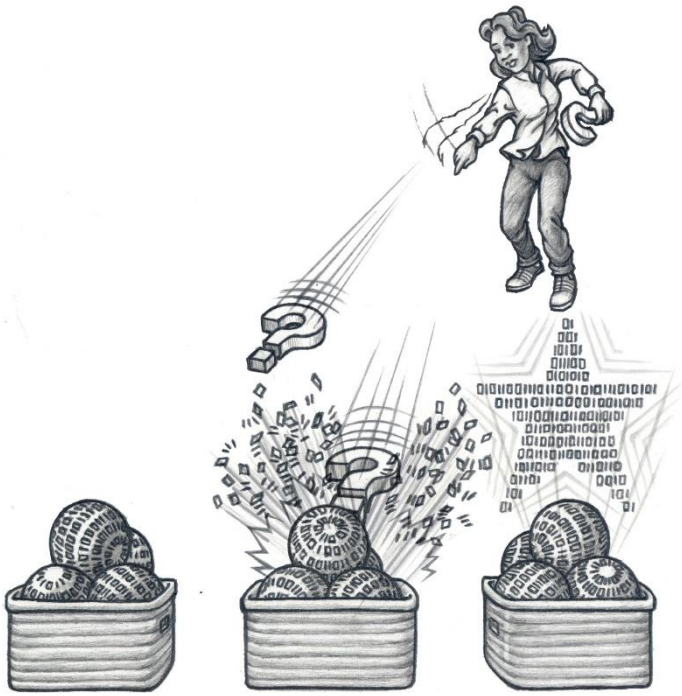

Fig. 20. A team member querying the data for the Queryable Quality pattern.

Figure 20 shows a team member querying the body of quality data, and receiving trustworthy results.

2.7.2 Context

Quality automation generates large amounts of detailed data on SUT behavior, but by default this is not easily accessible to the team as a whole due to the quantity of the data.

2.7.3 Problem

All the quality data must be available to those in the business with a need to know. The data must be accessible and available for viewing, drill-down from business-facing to technology-facing steps, and sophisticated query and analysis.

TODO see the diagram of hierarchical steps

2.7.4 Forces

Quality data must be easily available to the business for viewing, query, and analysis.

2.7.5 Solution

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

For quality data that does not need interaction, an information radiator (i.e., a large monitor to show data to the team) is a common tool that would supplement but not replace that purpose.

2.7.6 Consequences

Given the other patterns of MetaAutomation, or even an alternate implementation to address the quality automation problem space, this information is openly available for viewing, query, etc. within the company and governed as needed by role-based security.

This pattern completes a system of ultimate transparency within the business of what the SUT is doing in great detail and how fast it is doing it, obviating the common QA role of intermediary and interpreter for this information.

2.7.7 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 they happen, how often, is there an emergent pattern of failures there or is there a correlation with other failures, etc. People in the QA role use it to monitor app health as well as the health of their quality systems. Accountants doing work for Sarbanes-Oxley (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 is distributed across geographies or cultures, Queryable Quality represents 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 be added to the intranet site. Correlations can be studied between changes in app behavior or performance with the telemetry data, through the Queryable Quality site.

2.7.8 Counterforces

Queryable Quality would not be applied for a very small team, or a product where quality is not a priority.

Negative consequences of applying the pattern are just the overhead of implementing and maintaining an intranet site with view and query capabilities.

2.7.9 What This Pattern Depends On

Queryable Quality depends on Hierarchical Steps, Atomic Check, Parallel Run, Smart Retry, and Extension Check for creating copious yet detailed structured data on the product quality.

There is a powerful synergy here with the Hierarchical Steps pattern: the latter presents data on the check run from the root node first. The root node describes, at the highest level of abstraction, the entire check procedure. If the viewer wishes more information, the viewer can simply “drill down” in the intranet web page implementation of the Queryable Quality pattern, into the hierarchy to see more detail. The root of the hierarchy and steps near the root represent the business-facing and more general view of the check procedure (see Figure 2 in section 1.3) and drilling down to the leaf steps approaches the more technology-facing view that developers and people in the QA role may be concerned with. All of the data is available, but data overwhelm is avoided.

Queryable Quality depends on Smart Retry for managing any retries and recording and linking data for retries of a check.

Queryable Quality depends on Automated Triage because the quality notifications of an Automated Triage implementation include simple and query-enabled links to data on the intranet site that is the Queryable Quality

implementation for a software company. Automated Triage provides easy and germane entry points to the global and flexible view of quality provided by Queryable Quality.

2.7.10 What Depends on This Pattern

Automated Triage depends on Queryable Quality because most of the value of the notifications of Automated Triage is realized through following the links to the Queryable Quality portal.

2.7.11 How This Pattern Relates to the Pattern Language

This solution is an important part of the quality automation problem space that MetaAutomation addresses, because it enables data "pull" enabling the people of the business to view and do query and analysis on the data.

2.8 Extension Check

2.8.1 Description

Extension Check is about measuring and reporting on a functional requirement for the SUT that depends on non-deterministic internal conditions, for example, for highly distributed and interconnected systems that are difficult or impossible to drive directly.

Because the requirement concerned is non-deterministic, it cannot be driven directly by the code of a check, and therefore it is not suitable for measurement with the Atomic Check pattern of section 2.2. However, the data needed to measure it *is* available with appropriate hooks and product instrumentation, so the needed information can be persisted with the artifact of the original check and thereby made available for verification *after* the original check is complete.

Extension Check verifies the functional requirement based solely on the artifact data that results from such an instrumented run of an Atomic Check implementation.

The pattern name, with "Extension" modifying "Check," relates to five qualities of these checks:

1. They may run asynchronously to the original checks.
2. These checks do not drive the software system under test; they only analyze artifacts of check runs that *do* drive the system under test.
3. They are run at a lower priority relative to the original checks.
4. This pattern is only applied to quality measurements related to non-deterministic events in the business logic.
5. Because this pattern is applied to functional requirements that are not accessible to Atomic Check implementations, Extension Check represents an *extension* to the capabilities of MetaAutomation in the Quality Automation problem space.

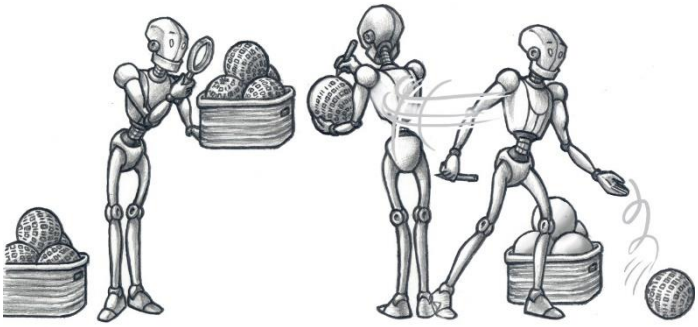

Fig. 21. Automation making checks on the artifacts of previous checks for the Extension Check pattern.

Figure 21 shows extension checks being done on the artifacts of atomic checks.

2.8.2 Context

Use Extension Check when a check has, in addition to the target verification (or verification cluster), some measurements, event-driven criteria etc. which can be stored in the check artifact data but for reasons of control and variance in behavior are not suitable for target criteria in separate product-driving checks.

2.8.3 Problem

Functional but non-deterministic quality criteria can be measured with automation, but not with a deterministic check verification.

Quality automation must measure and report on such quality criteria in a way that does not interfere with the primary and higher-priority quality measurements of the SUT.

2.8.4 Forces

The SUT has important quality criteria for which timing is non-deterministic or externally decided.

2.8.5 Solution

The solution is to measure and include data related to the non-deterministic quality criteria in the artifacts of check runs that include the software unit(s) where those quality criteria are at issue, but not fail the product-driving check based on any condition of the non-deterministic criteria.

Since the artifacts of those check runs have all the data needed to determine whether the non-deterministic quality criteria cause an actionable condition or not, the Extension Check pattern is applied to run analysis on the artifact data and fail the extension check if an actionable failure condition exists. This way, the non-deterministic quality criteria is still measured and appropriately actionable, but the check that drives the SUT and collects the data is not blocked on a failure in the non-deterministic quality criteria.

2.8.6 Consequences

This pattern enables reporting, communications, and action on non-deterministic quality criteria for the product, in a way that does not block other quality measurements nor interfere significantly with performance of the check run.

2.8.7 Examples

Extension Check describes, for the quality automation domain, a way of deriving value in measuring and reporting quality from data from experiments that cannot be controlled.

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 is done with data from "experiments" in the remote universe that humans do not have the power to initiate 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 served in response to end-user activities, balances, etc. and of course asynchronously while the activities are being done. The new-ad event is given 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.

Analysis is done after the check run to determine if the ads are correct or not, with acceptable timing and other criteria.

2.8.8 Counterforces

Extension Check would not be applied if every aspect of functional quality is driven directly by events under the product team's control, or easy to synchronize without changing the measurement result.

A negative consequence of applying this pattern is the cost of implementing it and integrating with the rest of the quality automation system.

2.8.9 What This Pattern Depends On

This pattern depends on Atomic Check and Hierarchical Steps for short, fast checks with results in a coherent data structure that provides placeholders for any product instrumentation, inline or for the whole check.

2.8.10 What Depends on This Pattern

Extension Check results deliver actionable data to Automated Triage for sending notifications, and Queryable Quality for query and analyses.

2.8.11 How This Pattern Relates to the Pattern Language

This pattern addresses verifications of requirements that are non-deterministic or have timing that can't reasonably be controlled, within the quality automation problem space that MetaAutomation addresses.

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.

I suspect that one reason that the best positive approach has been neglected until now is that, as inspired by Myers' book [Myers 1979], people tend to focus exclusively on finding bugs; bugs are more concrete, easier to understand and simpler to measure. What was lacking was a clearly expressed vision for value to the software business of an optimal automation-based approach to measuring and reporting on quality for the SUT, to be used *in conjunction with* the search for bugs. I try to provide that vision with this paper and related writings.

MetaAutomation is extensible. Patterns to be added in future may fill out the quality automation problem space specifically to address machine learning in software.

4. ACKNOWLEDGEMENTS

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

[Meszaros 2007a] Meszaros, Gerard, “xUnit Test Patterns,” 2007, p. 454
[Myers 1979] Myers, Glenford, “The Art of Software Testing,” 1979, p. 5-8
[Sebillotte 1988] Sebillotte, Suzanne, “Hierarchical planning as method for task analysis: the example of office task analysis” published in Behaviour & Information Technology, 7:3, 275-293, DOI:10.1080/01449298808901878, 1988
[Cessna 1984] 1967 Model 172 And Skyhawk Owner’s Manual, Version copyright 1984
[Goucher 2009] Adam Goucher’s blog is here http://adam.goucher.ca/?cat=3
[Meszaros 2007b] Meszaros, p. 359
[Roseberry 2017a] Wayne Roseberry, personal communication
[Roseberry 2017b] Ibid.