MetaAutomation: A Pattern Language to Bridge from Automation to Team Actions for Quality

Matt Griscom

matt@wisewillow.org

Abstract

Regression testing automation provides an important measure of product quality and can keep the quality moving forward during the SDLC[1]. Unfortunately, automation can take a long time to run, and automation failures generally must be debugged and triaged by the test automation team before any action item can be considered or communicated to the broader team. The resulting time lag and uncertainty greatly reduces the value of the automation, and increases time cost and quality risk.

MetaAutomation is a language of five patterns that provides guidance to new and existing automation efforts, supplies fast and reliable regression testing of expected business behavior for a software solution, speeds quality communication around the team, and reduces latency and resource cost.

The five patterns, Atomic Check, User Pool, Parallel Run, Smart Retry and Automated Triage, form a sequence, representing an order in which the patterns would apply, and a network of dependencies from the more dependent to the less dependent patterns.

For an existing automation project, the least dependent pattern, Atomic Check, can be applied in whole or in part to run the automation faster and create results which are more actionable. If enough of Atomic Check is followed, the dependent patterns can then be applied to further speed, direct and enhance the value of communications resulting from the automation.

The patterns are language-independent. The talk that presents this paper provides and demonstrates a platform-independent sample implementation of the Atomic Check pattern.

Biography

Matt Griscom has 20 years' experience creating software including test automation, harnesses and frameworks. Two degrees in physics[2] primed him to seek the big picture in any setting. This comprehensive vision periodically puts Matt in the vanguard. Matt loves helping people solve problems with computers and IT.

[1] Software Development Life Cycle
[2] BA in physics from Wesleyan University, BS in Astronomy from University of Washington

1 Introduction

In the worlds of Test and QA, automation is about making a software product do stuff. Automation authors answer the question "Have you automated this test?" by making the product do steps of the test automatically. They might add on a verification or two, especially if the test is on an API or service rather than a GUI. If the automation fails, then the automation author must manually determine what is not working, usually by reproducing the failure and stepping through the code. Until then, the test failure is not actionable, other than a vague warning to the team that something is not working. It might be serious, but they do not know yet. In the meantime, part of the product quality goes unmeasured because a test case that would measure it is marked as "automated," and while the automation itself is failing to deliver value for the bit of target quality, manual testers focus their efforts elsewhere. N pieces of automation that fail for an unknown reason thus represent N items of product quality going unmeasured for a time that scales with N, causing project risk that scales as N *squared*.

MetaAutomation includes and enriches automation itself, and it offers a higher-level perspective, a much more expansive horizon. Using techniques of the patterns of MetaAutomation, verification of correct product behaviors will run faster, scale better, and deliver important information *with no deciphering or wait time needed*. The team-wide quality process becomes more robust and interactive, delivering better quality information and making sure that the business logic of your application is solid, so the quality of your software project can always go forward, never backward.

This paper presents the first pattern in the language, Atomic Check.

1.1 Why a Pattern Language?

The book *Design Patterns*[3] is the most famous of books describing a set of meta-solutions about common problems in software engineering. As the preface states,

> *Design patterns capture solutions that have developed and evolved over time*.[4]

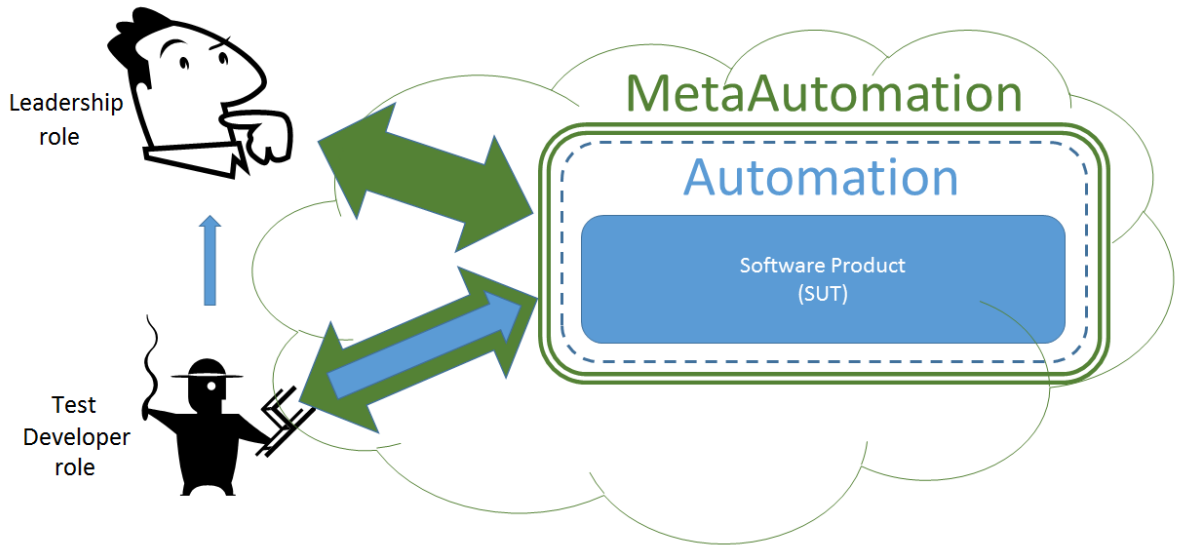
The book's simple and elegant solutions don't tell you what language to use or what you should name your classes, but they are extremely useful high-level answers to the questions that software developers face when they are choosing designs to build or refactor software systems or subsystems.

MetaAutomation presents a set of five patterns that address turning product automation into action items and decision points. The patterns have components that many teams have implemented with success, and some new properties that bind them together for the greatest complementary strength. There is a strong and ordered dependency between the patterns, and as a group, they address important issues. For example: If automation is about running the product through its paces in a controlled environment, how does this deliver quality information to decision-makers on the product development team?

MetaAutomation is more than a set of patterns; it is a pattern *language* with dependent and interdependent patterns to address the problem space including software product automation on one side, and decision-makers and decision-points on quality on the other.

The following diagram expresses the relationship between traditional, basic automation and MetaAutomation, with the arrows to represent channels of communication and information flow: the blue arrow for traditional automation, and the green arrow to show what MetaAutomation can do.

[3] Gamma et al., 1977
[4] Ibid, p. xi



This figure shows the relationship between traditional automation and MetaAutomation. One of the benefits of this book is disintermediation: with MetaAutomation, the test developer no longer needs to interpret the results of test automation for others on the team.

1.2 Testing vs. Checking

Given that software testing is about measuring, communicating and promoting quality, leadership often sees automation – that is, making your software product do things automatically – as a way of doing all of the above *faster*. Unfortunately, it does not work that way.

People are smart, but computers and computing power are *not* smart. People running user stories, test cases or doing exploratory testing are very good at finding large numbers of bugs, within the limits of attention, getting tired, bored, etc. People are great at spotting things that are not as they should be, for example, a flicker in an icon over here or a misalignment of a table over there, or a problem with discoverability.[5]

Automated product testing, done well, has huge value. Automation is excellent at preventing regression of product quality issues quickly, repeatedly, and tirelessly. Automation does not get tired or bored. Computers are very good at processing numbers and repeating procedures, and doing them fast and reliably, and doing it at times like 3:00 AM local when your team members are home sleeping.

However, automation is *not* good at finding product bugs or anomalous issues like the flickering icon. You need good human testers for that.

Atomic Check is the first of five patterns in the language. It describes ways to do automation well and effectively, run faster and with better scale potential than traditional automation, and with actionable results. The artifacts from following Atomic Check, the detailed, strongly typed, hierarchical results of a

[5] "Discoverability" answers the question: Can an end user discover how to use the product, using only cues from within the product itself (excluding help documents)?

check run are so powerful, they open the quality process to faster, directed and more focused communications with all the data for quick follow-up or even correlation with existing bugs.

This paper uses the term “checking,” proposed by James Bach et al.[6] to define automation that drives tests. A single automated procedure that measures a defined aspect of quality for the SUT is a “check.” The term “check” applies where more commonly a professional in the space might use the term “automated test” but since testing is an intelligent activity done by humans, the term “automated test” becomes an oxymoron; once a test is automated, it is no longer a test in the same sense. If done well, it is fast, reliable, tireless and highly repeatable, but the value is very different from the same procedure run manually by a testing professional.

If “testing” generally is about measuring the quality of the product, then “checking” is a very important, specialized subset of testing. Designing and developing checks takes significant testing skill.

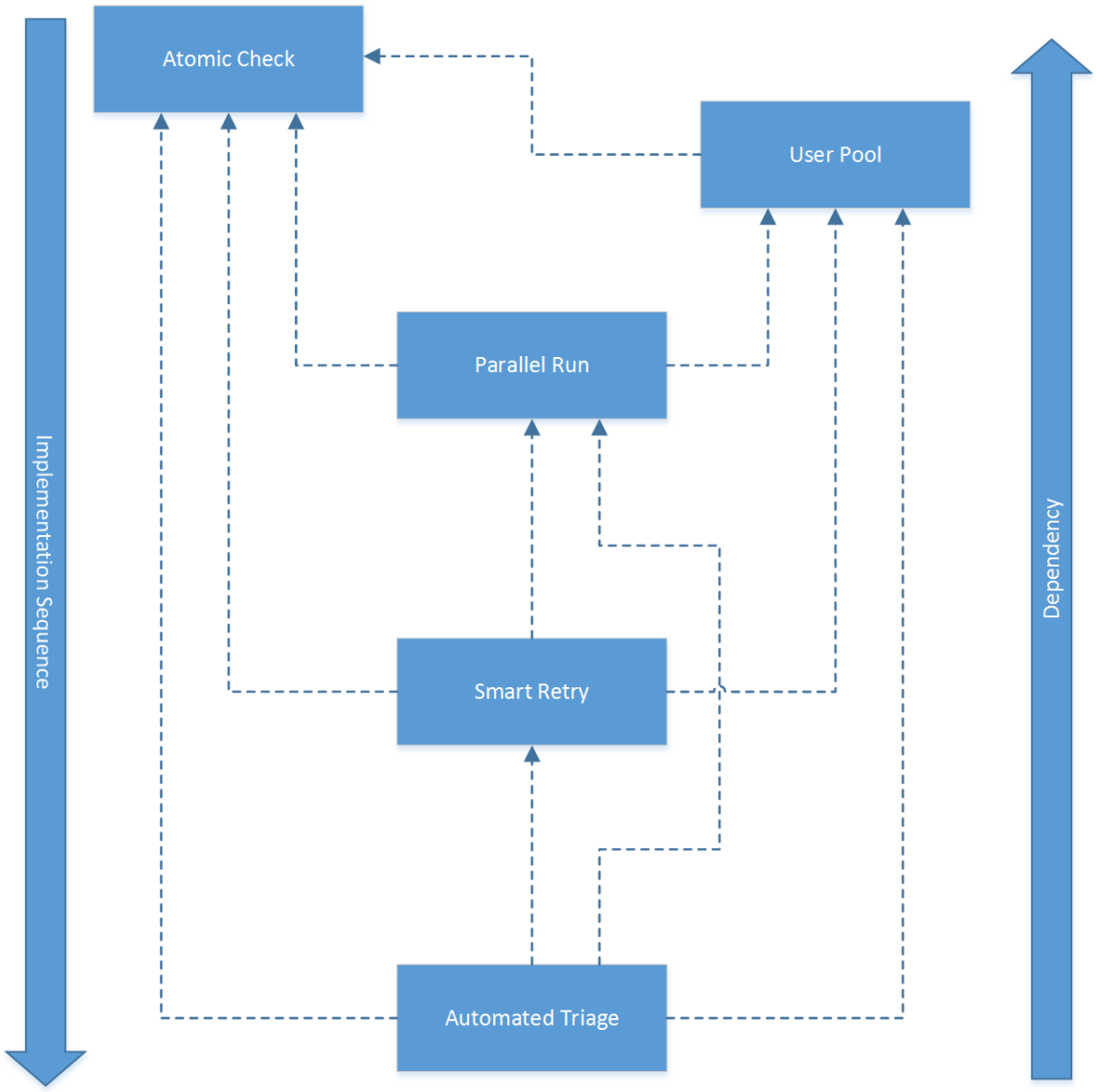
The term “check” emphasizes an important reality of developing modern, impactful software: the team needs both manual testing and automation. One does not substitute for the other. However, if the checks are well defined and implemented, and they are passing, they obviate manual checking of the targets of the checks.

The name of the pattern described in this paper, Atomic Check, reflects the shift away from using the term “test” to reflect automation. However, this paper uses both “test” and the deprecated term “automated test” because they are still common in the vernacular of the industry.

1.3 Relationships of Atomic Check to Dependent Patterns

Due to limited time and space, this paper and the associated talk focus on the first pattern in the MetaAutomation pattern language: Atomic Check. The following diagram illustrates the sequence and dependency network between all five patterns.

[6] Bach et al., “Testing and Checking Refined”



1.4 Business Continuity

There are many reasons for which your business might suddenly find itself without the team member who is most knowledgeable of the automation project. This is a well-understood risk; what if it happens to you and your team?

Applying the Atomic Check pattern, along with the four other patterns of MetaAutomation, reduces your risk in these ways:

- Checks are as short and as simple as possible, minimizing maintenance time by reducing the number of potential failure points and reducing the length of time a check run requires.
- Check failures are highly actionable, reducing dependency on automation authors as intermediaries.

- Automation should use the same language as the business logic of the product where possible. This greatly reduces the mystery associated with the finer points of non-standard languages and assures that there is in-house expertise if needed.

- Coding the automation with an IDE and in-house code, just as the software product does, is a natural approach. This lowers the incidence of blocking behavior limitations or bugs that cause the team to wait on external factors. Mysteries can be resolved, if needed, by an investigation or debugging session by experienced in-house talent.

- 3rd-party test automation tools are probably unneeded. This reduces licensing costs and/or external dependencies which can cause mysterious failures or external blocking bugs. Mysteries can be resolved, if needed, by an investigation or debugging session by experienced in-house talent.

- Self-commenting coding techniques as part of creating actionable failures for the pattern. This creates readable code and largely eliminates the need for traditional code comments which can become misleading or incorrect.

- Applying the full pattern language allows all interested team members to interact with the automation, which distributes knowledge of the tools and capabilities more evenly around the team.

2. Atomic Check

The Atomic Check pattern describes how to design and write a check that measures a simple and important aspect of product behavior. The pattern is a guide to maximizing the following values:

- The actionability of a check failure, so that decision makers will know what to do about it
- Scaling the check run across machines, virtual machines or operating system instances
- The speed of verifying the business behavior

While minimizing these aspects:

- The number of debug sessions required before a given check failure becomes actionable by any team other than the one that owns the check code
- The duration and scope of any part of product quality going unmeasured due to a check failure
- Latency in communicating product quality issues

Checks following the Atomic Check pattern are end-to-end and may include integration, regression or performance checks.

2.1 Basic Requirements of Atomic Check

The Atomic Check pattern requires that each check verify just one thing about product behavior, plus any steps or product behaviors that are required to support the verification that is the focus of the check.

Each check must be as repeatable as possible in how it drives the SUT. There is no place for random, pseudo-random or quasi-random values.[7]

Each check is designed to be independent of all of the other checks. Checks can be run singly or in sets in any order, in series or parallel.

"Fail soon, fail fast" summarizes these requirements of the Atomic Check pattern:

- Detect a failure condition for the check as soon as possible. This is important for root cause. For any statement or line of code in the check that could fail, if earlier or simpler statement could detect the failure and report it, then there must be such a statement in the code. For example, your code could assert that an object is not null or that it can be found, with an informative message formed and reported for the case of assert failure.
- Make the check fail as fast as possible on detecting a failure condition. The most common pattern for making this happen is to create and throw an exception that fails the check. For example, the Assert idiom does this by throwing an exception.

The check must fail with detail and accuracy. All check failures need to be traceable to their root cause, at least in the context of the check code or harness.

The check logs – A.K.A. flight-recorder logs of what the check is doing - must be separate from the record of failure for many reasons, starting with the fact that if the logs are useful at all, they are only useful at check development time. Logs do not otherwise get stored past the check run, but in case of check failure, the harness running the check will archive the artifact of the check run for later perusal and automated triage.

Just from these basic requirements, it follows that Atomic Check requires either deviation from product coding styles and requirements, or another set of coding styles and recommendations.

2.2 Automation is for Checks, rather than Tests

Automation for measuring product quality is indispensable for an impactful software product. Without it, the quality risks are too great:

- A business logic failure or incorrect behavior can impact the workflow of the team.
- An incorrect behavior can cause additional downstream bugs, as developers "work around" the failure.
- The development team can get distracted (A.K.A. "randomized") by the need to go back and fix an issue that was introduced some time before.
- If the issue impacts a customer before the team is aware of it, crisis management can take over to fix the issue, with its own dissonance, costs, risks and delays.

Checks, written and run well, will notify the team quickly of quality regressions, thus avoiding these and other risks.

However, one limitation with automation is that, other than the default point of verification (did the application *not* crash?), the only points of verification are what is written in the code, implicitly or explicitly. Most points of verification must be written explicitly, with reporting that describes root cause for the failure case.

[7] Testers use these sometimes in an attempt to increase test coverage, but are really only suitable to fuzz testing, and even then, the random values must be persisted somehow to make the tests repeatable.

The term “check” reflects the target verification of the automated test. For example, if the team automates a verification that a user can change the preferred color scheme for a client app and that this change is persisted successfully, then this piece of automation “checks” that the change persists. The preliminary steps – logging in, going to a settings page, changing the preference etc. – have preliminary assertions.[8]

2.3 Check Setup and Teardown

Modern automated test frameworks provide for a test setup (initialization) and a test teardown (clean up), separate from the main body of the test. If these methods are implemented, they run just before and after the main test, and on the same thread.

Atomic Check does *not* allow these methods because analysis of a check failure would be difficult or impossible if there was a failure in the test setup or test teardown. Atomic Check only details and persists failure in the scope of the check method, not in setup or teardown. Using a class setup (a setup for a set of checks) might also interfere with the ability of checks to run independently of each other on different client processes.

Move all setup steps in line with the main procedure of the check. Setup steps are treated just like any other preliminary step of the check, and the artifacts related to check failure are treated the same way; they may be actionable bits of information there for people or automated processes to follow up on, to analyze patterns of failure or just get the checks running again.

Using a setup method pattern would make it impossible to apply the Smart Retry pattern later. Given that a check failure happens with an exception thrown out of the check method scope, a failure in a setup method cannot generate artifacts in the same system that will enable root cause to be determine. The check would fail, but the artifacts would be missing failure cause information.

One could argue that the historical reason for using the synchronous setup idiom is that a failure “aborts” the automation, rather than fails it. The reason to have an “abort” is that it clearly signals that the failure is not an action item for the product developer team or role; the automation author or authors must fix it. However, the team no longer needs this signal: with the Atomic Check pattern, the artifact answers the question of who needs to follow up on a failure (if anybody). With complete MetaAutomation, if there were an action item for a person on that team, that person would receive notification (e.g., by email) with a link to an intranet site with all of the information needed.

Another historical reason for the setup idiom might be the idea that using setup and teardown methods will ensure that a failure of the main check method always has failure in the SUT as root cause. This would simplify triage; the idea is that if the main check method fails, it is clear that the product developer has to fix it. This reason simply is not valid anymore; with modern software applications, dependencies are much more complex than they used to be. For end-to-end tests, generally there are many factors outside the SUT, which might cause a check to fail in the main method of the check.

If your automation requires a significant clean-up after a check where any issues that might happen during the clean-up are *not* related to the SUT, consider using the User Pool pattern or other asynchronous method if that can free up the thread (and associated resources) more quickly.

In summary, although many automated test frameworks offer setup and teardown methods, the historical reason for offering these methods is no longer valid, and the Atomic Check pattern requires that you not use them.

[8] See section 2.8.1

2.4 Outputs from a Check

The runtime process of MetaAutomation starts with what information comes from a check.

If the automation drives an application GUI, it is possible that a knowledgeable tester staring at the screen during the entire automation run can get some useful information out of the automation run. (I have seen this done.) However, this is very expensive, error-prone and risky, and any information coming out is not likely to be of very high quality or complete.

For an application GUI, the harness can take screenshots, and for some specific applications, these can be useful, but automated parsing of screenshots into something actionable is something I have never seen done successfully.

An important part of MetaAutomation is focusing the information from a check on maximizing value for the consumer of this information. Consumers, i.e. customers, of this information fall neatly into two roles: automated test developers who are in the process of developing or maintaining the checks, and decision-makers[9] who receive the information resulting from the check run.

The Atomic Check pattern focuses information flow and value for the different needs of these two roles.

Everybody who has an interest in quality information about the SUT from a check will want to view and act on the artifacts, i.e., the results of a run of that check. This information includes whether the check passed or failed, and in case of failure, enough information to identify the root cause of the failure.[10]

At check development time, the developers of the automation might need more details from the procedure as it is executed, i.e., what steps are completed and when. This "flight-recorder" log helps correlate automation code and what is happening as it drives the product. Stepping through code in an IDE can provide this information, but for a non-trivial check the steps are useful in the artifacts should the check fail. The sample project provided gives an example implementation for this purpose and supports check steps in a hierarchical dependency of any depth.

Logs and artifacts are two distinct streams of information, as shown in the following table:

	Logs	**Artifacts**
Creation time	The harness streams logs as the check runs.	The harness persists artifacts at check completion.
Source of the data	Logs stream actions that the check performs as it drives the SUT.	Artifacts hold information on the check, plus failure and specific detailed context to express root cause for a failed check.
Role of the data customer	Logs can be used by automation authors to manually follow up on failures.	Artifacts are focused and concise enough to present with an XSL stylesheet to any decision-makers on aspects of product quality, and they are

[9] E.g., managers, leads, and product developers

[10] Please see section 2.6 for more information

		strong enough for automated parsing and comparisons.
How the data is presented	This is flat but usually readable text in a log file or an output window in an IDE.	Artifacts are data only, no inline presentation.
How the data is persisted	This data is not used after a development session is over, so can be discarded.	This data is persisted at least for the duration of the SDLC, possibly longer.
How the data can be used	People create logs to be readable by people, but they tend to fill with lots of information that is not useful.	Artifacts support fast and robust automated triage and analysis, and an XSL stylesheet can present the data very nicely.
The importance of the data	This can be an aid for developers at test development time, but it is not required.	This is a basic requirement for MetaAutomation, so without this data, the value of test automation to your software project is limited.

In case of a check failure, in order to make the failure actionable, the stage of the check that failed is useful information that will help in failure analysis (whether automated or not) or correlating check failures. However, in order to be useful for humans and automated processes (as seen with the other MetaAutomation patterns) to triage, it is better to leave out the flight-recorder log of every step that the check performed in order to get to the point of failure; that quickly gets too verbose for automated parsing. For applying the Smart Retry and/or the Automated Triage patterns, it is especially important to keep information about the failure concise; unnecessary information might cause either of these patterns to fail. To support automated parsing and succinct logging for a complex check, consider expressing sections of the check procedure in hierarchical steps[11].

To support long-term storage of the check result artifacts and automated parsing, querying and comparison, the Atomic Check pattern considers logs and artifacts to be two separate entities.

2.5 Logs

The logs from a check are useful at check development time and maintenance time. The Atomic Check pattern helps create simple checks that minimize maintenance time; however, product changes that require changes to the check are still possible. For a complex check, logs might help at development time.

[11] The sample implementation demonstrates support for hierarchical steps that go into the structured artifact in case of check failure.

Because logs are streams of text, they tend to be verbose and difficult, expensive or impossible to parse reliably. The Atomic Check pattern therefore recommends that they be discarded at completion of any check run.

The working sample of an Atomic Check implementation provides for hierarchical check steps that output to a log *and* are stored in a structure in the artifact in case of check failure. The check developer can use steps like these instead of simple log statements, and it will help at both check development time and at artifact triage and analysis time.

Atomic Check recommends discarding simple log streams when the debug or development session is finished.

2.6 Artifacts

Artifacts are the check results that are stored for some length of time for later review, manual analysis or automated analysis. They record the name or unique identifier for the check, the "pass" or "fail," start time and duration, but much more than that as well. In case of failure, the harness records the point of failure with enough information to identify root cause.

Running many checks, and running them frequently, generates large numbers of artifacts for the check result data. Keeping that data compact allows you store more of it and do more subsequent analysis on the data. The sample project provided uses XML.

In the case of storing the artifact data in a flat file system, XML is ideal because it is:

- Text-based
- Compact[12]
- Extremely scalable
- A well-known and well-supported W3C standard

With XML element names and attributes, the data is automatically future-proof and can be extended as needed without loss of information or risk of data being unrecognized due to schema changes.

XML is also a W3C[13] standard, unlike JSON, and has the standard presentation language XSL to make the artifact data accessible for human viewers.

Mostly, the data that goes into the log for use at development time is different from what the harness saves as an artifact because the uses and purposes of the data are so different. However, there is one bit of information that serves well in both destinations: what the step (or hierarchical steps) of the check are at point of failure.

2.7 Dependent Operations of an Atomic Check Implementation

The focus of a check that follows the pattern is an atomic verification, which usually consists of one or more assertions. That is not all that the check does though, because in any dynamic system there will be dependent operations.

[12] And compressible

[13] The World Wide Web Consortium

For example, if we are testing a REST service for the ability of a certain account to log in, the focus of the check is successful login, according to the criteria we specify: the message received *and* a given session cookie. The message and the cookie are both requirements for success.

The dependent operations in that case include:

- Test infrastructure
- The client opening a network connection to the service
- The client creating, serializing and sending an HTTP request
- The server successfully generating a response
- The response has a compatible HTTP status (probably a 200 OK)
- The response body is successfully deserialized

The check includes assertions which might be implicit, that is, not directly expressed in the code. Check developers might choose to make the assertions explicit to ensure that in case of failure at a given dependent operation, the artifacts are informative enough to identify root cause in the context of the check run. An alternative is to supplement the artifact related to a check failure with server-side log data around the failure, to help determine or disambiguate root cause.

The Atomic Check pattern still applies, though, because the focus of the check is still one thing: successful login. Testing that one thing cannot be done in a simpler way or with fewer potential failure points.

2.8 Points of Verification for Atomic Checks

The target of the check probably requires some steps to get there. Most such steps are prone to failure - or, if you take potential coding errors in the automation code into account, all of the steps could potentially fail.

The term “assertion” is useful here because it refers to all assertions that happen in the check, including ones that could be missed on casual code review, for example, null-reference exceptions (null-pointer exceptions for a native runtime) or required properties that do not default. The assertions that aren’t visible in code that is owned by the team or the project doing software checks are “implicit” assertions; they aren’t coded for explicitly in code authored by that team or project.

Preliminary steps lead up to the target of the check. For example, opening a network connection (or some higher-level object that abstracts this out) is probably a preliminary step. The target of the check verifies a piece of important business behavior, and is the purpose for which the check exists.

2.8.1 Preliminary Assertions

On check setup or initialization, please see section 2.3 on Atomic Check Setup and Teardown.

There are two types of preliminary assertions:

Implicit preliminary assertions are part of the preliminary check steps and do not require any extra code from the check author, and explicit preliminary assertions that throw an exception with a failure message on check failure.

For example, if a network connection made as a preliminary step times out, then the network timeout exception that fails the check is a result of an implicit assertion, i.e., that the network connection can be made and does not time out.

Explicit preliminary assertions are assertions that help the check fail faster than it would otherwise, and with better information.

For a simple example, if a preliminary step requires creating an object of type User, and if the User object creation might fail returning null, the check would fail by default at the following step, which did something with that User object. In this default case, though, it would fail with a null reference exception or something similar (depending on your programming language and framework). This failure of the check could even be ambiguous, too, with only that information, and therefore not actionable because root cause is not clear. This would require follow-up with a test-automation-and-product debug session by the check author or someone else on his/her team.

To implement the Atomic Check pattern, this example needs at least one explicit preliminary assertion. The check asserts (or uses equivalent test logic) that the User object is not null, with informative error information about that to throw with the exception, including the User object name or ID and any other useful information.

The check requires another preliminary assertion if the User object has some property that is required for the check to succeed. For example, if the check would fail later in case the User object is not currently active, then put in an explicit assertion for that property, with error information.

Explicit preliminary assertions must never cause the check to fail if the check were not going to fail anyway at a later step. However, if the check is in a state of imminent failure, and this is measurable in the context of the code, Atomic Check requires adding an explicit preliminary assertion to fail the check faster and with a more actionable message. Otherwise, there is only the implicit preliminary assertion, which might fail occasionally with neither information closest to the root cause of failure nor an informative message with the exception object. In the latter case, the team might have to wait on the automation author to follow up, try to reproduce the failure, and find root cause before it is determined whether an action item exists for anybody other than that author or whether there is some unmeasured part of product quality.

The case where a check might fail with incomplete information for the customers of the check automation is exactly the one where the check needs an explicit preliminary assertion. An explicit preliminary verification helps the check fail sooner (i.e., faster, in the scope of running many such checks) and with more specific information to help speed communication around your Atomic Check implementation and make the dependent patterns of MetaAutomation possible.

2.8.2 Target Points of Verification

Each check following the Atomic Check pattern has one target verification. A target verification answers the question, “Has the SUT passed the check?”

This might just be the final assertion of the check, but may also be a little more complex. Maybe it has to be two assertions rolled into one, depending on the bigger picture into which your check fits.

At check design time, you will have to ask this related question: “For the target behavior of this check, and from the point of view of clients of this behavior, what does success mean?”

The target verification might be just one assertion, or it could be two or more.

Two assertions - call them A and B - cluster together as part of the target verification of an atomic check if they meet all of the following criteria:

1. A and B are both important to business behavior (or, client needs)
2. They are *not* dependent on each other in terms of the verification steps
3. They *are* logically linked together. If A should fail, the state of B at check failure time must be recorded in the artifact as well because this affects the action item that might result.
4. Between the measurement of A and the measurement of B, there is no interaction with the product or any external entity over the network (and hence no transient failures that might be eliminated from the check result summary with the Smart Retry pattern).

The clients of the subject behavior of the target verification would need the behavior in order to continue a scenario, beyond the scope of the check. Therefore, an assertion on something that is not required to continue the scenario (if only hypothetically) beyond the scope of the check is *not* a good candidate to include in the set of assertions in the target verification.

The Atomic Check minimizes parts of the product that go untested. This means that the target verification might require reporting more than just the result of a simple Boolean verification, even while at the high level, the result of the check is Boolean: pass or fail. The following example clarifies this:

Suppose it is a real estate web site, and you want to automate a check that verifies that search results on the public-facing site will show properties for sale. The Atomic Check pattern requires that the check be as simple as possible and that it verifies something about the site that is a prerequisite for customers to proceed.

The check has these steps:

- Start a web browser with the URL of the site
- Click as needed to get to the property search page
- Change criteria as needed to get search results
- Verify that at least one real estate property is shown for sale

The target verification includes ensuring that the page shows at least one real estate property.

When implementing this check, a developer needs much more detail to make it happen. People are smart, but computers aren't, so to get from the people-friendly four steps listed above to something that the automation needs, some steps are added (in italics):

1. Start a web browser with the URL of the site
2. Click as needed to get to the property search page *translates to*
    a. *Find the menu item on the page that shows the choices needed to proceed*
    b. *Click the menu item*
    c. *Find the "Property Search" submenu item*
    d. *Click the submenu item*
    e. *Wait for the Property Search page to load*
3. Change criteria as needed to get search results *translates to*

    a. *… (maybe nothing is needed to be done here)*

4. Verify that at least one real estate property is shown *translates to*
    a. *Find the HTML item that shows the number of properties that result from the query*
    b. *Read the InnerText of that HTML item*
    c. *Parse the text for the desired number*
    d. *Assert that the number is greater than zero*

Any one of these steps could fail, due to check dependency failures, check code failures, or of course failures from the SUT.

One of the goals of the Atomic Check pattern is to minimize the need for testers to follow up on a failed check with an intensive debug session just to answer the question "What happened to cause the failure? Is this a test or a product bug, or something else?" Debug sessions on failed checks are very expensive and time-consuming, and as a result, they might not even happen at all. Even if they do happen, they cause a delay and some disorder in the productivity of the team. The result is reduced value in the checks: they can still sometimes verify correct product behavior, but the team will tend to view failed checks as a test-owned failure rather than a product issue, and the manual debug session that might turn a failed check into a fixed product bug is a barrier to the product team.

Therefore, it is important to ensure that the artifacts from a failed check contain the information needed to know *if* someone needs to follow up, or at least maximize the chances that the artifacts are sufficient that the importance and impact of the failure is known, without a manual debug session.

Suppose, for example, that with our check on the real estate site, the check failed at step 4c: the parse failed for some unknown reason. If the target verification is step 4d, then the target verification did not happen. It appears that an implicit preliminary assertion failed, and the artifacts that result from this check failure include a format exception and a stack trace that points to a file and line number in the source code. With that information alone, there's no way of knowing whether the root cause of the failure comes from the SUT or the check code, so the check result will be ignored until and if somebody on the test team can follow up. If there is a product bug, it is still undiscovered, although if there is a pattern of this check succeeding before, that change in the pattern of success is significant.

For this check on a real-estate web site, if step 4d – verifying that there is at least one property returned in the ultimate search – succeeds, then that is sufficient to pass the check. The end-user could continue in theory from that point with a further scenario because there is at least one property to look at. Step 4d is the target verification.

Consider the relationship between steps 4a, 4b, and 4c; if step 4a fails, step 4b is meaningless, and if 4b fails, 4c is meaningless. It is clear that steps 4a, b, and c are preliminary steps, along with any part of steps 1-3.

However, the implicit preliminary assertions of those steps as written is not sufficient. Back to the example of a failure at step 4c: if the check fails with an exception from trying to parse the text, there still is not nearly enough information to decide whether the check failure is actionable or who needs to take what action.

Suppose the text value is "four," and that is consistent with design of the product. The failure to parse that string into an integer would then be a bug with the check code.

Suppose the text value is a zero-length string. That might be a significant and actionable product bug, but the check result does not show that, so the check result itself is not actionable and the product bug is unknown for now.

This check needs explicit assertions so that enough information goes into the artifact associated with the failure, such that the results of the check are actionable. Another way of looking at this issue is that, when there are a large number of check results and many failures, it is important that the check artifacts discriminate between the zero-length-string issue and the "four" issue above.

To cover the possibilities for steps 4b and 4c:

Add an explicit assertion to step 4b that the inner text of the HTML element returns a string with greater than zero length. This may be browser, or framework-dependent, but it is probably safe that accessing the inner text of an HTML tag will not by itself fail, e.g., with a null-reference exception.

Add an explicit assertion to step 4c so that if the parse should fail, the harness reports this as part of the check run artifact with the actual string. We need to know if the string that is expected to contain the integer we want is "four" (which would be a check bug, if the product design allows for that kind of behavior for the SUT) or perhaps "1,0,01" (which would represent a product bug).

Consider again the example from above: verifying successful login over a REST service. The final conditions for a successful login include:

1. The HTTP status of 200 OK
2. Verification that the session cookie is the correct type
3. Verification that the body of the response includes the expected indication of login success

The Atomic Check pattern allows a set of grouped target assertions if all these conditions are true:

1. This is no interaction with the SUT or any outside system that could introduce latency or another point of failure after the first assertion in the set and before the last.
2. There is a logic reason related to functionality of the product to group the assertions together.

In case of check failure resulting from any assertion in the set, the failure report must include the result of all assertions in the set. This is part of what makes the check atomic: assertions in the set fit together as one, and in case of failure, the harness reports all product information resulting from that assertion set, no matter the results of any individual assertion in that set.

This type of login verification can still be an atomic check, even if it has multi-part assertions at the check target - that is, that login is successful. But, the correct cookie and the login success status both depend on the response coming back with an HTTP status of 200 "OK"; any other status is not compatible with a successful login, and in addition, the HTTP status is (or should be) measured before an attempt is made to stream the body of the response off the server. Therefore, the verification of HTTP response status of 200 must be a separate assertion. The cookie and the response message, though, meet the criteria for a set of grouped assertions in the target verification.

Therefore, the final assertion set of the check includes a check on the login cookie and a check on the body of the response. The cookie assertion and the body assertion must both be done, and then the check fails if either or both of these checks fail, and in any of these failure cases, the report generated at check failure includes information about both assertions in the set.

The focus of this check is the final assertion set, which answers the question: was the login successful and correctly formed according to the designed product behavior? Given that there are two Boolean assertions in the set, there are four possible outcomes:

1. Login was successful and correctly formed, so the result for this positive check run is “true.”
2. Login was successful according to the body of the response, but the cookie is absent or does not meet some criterion for success. The check run result is “false,” and the artifacts include this information from the target verification:
    a. That the body was as expected
    b. That the cookie did not meet the success criterion
    c. Information about the cookie including details on why the criterion were not met
3. Login was not successful according to the body of the response, but the cookie looks correct. The check run result is “false” and the check run artifact includes this information:
    a. The fact that the body did not meet success criteria
    b. Details on how the body did not meet criteria
    c. The fact that the cookie met the success criteria
4. Login was not successful, and the body and cookie were not as expected for the run of this positive test. The check run result is “fail” and the artifacts include all of the information about the assertion results in that final verification set.
    a. The fact that the body did not meet success criteria
    b. Details on how the body did not meet criteria
    c. That the cookie did not meet the success criterion
    d. Information about the cookie including details on why the criterion were not met

In case of check outcomes 2, 3, or 4, the code for the target verification places all of the information described into the exception and thrown to fail the check. In case of check fail, the artifact generated for this check run would contain information on the state of the cookie and of the response body. For a check that verifies login details for a custom REST service, analysis needs all of this information to describe the login failure and follow up at triage and analysis time or at runtime for the Smart Retry pattern.

In addition, if the check code and harness implement hierarchical steps as with the sample project, a failed check will include hierarchical steps leading up to the failure to clarify the context of the check failure event.

Given that the server side of the login is also part of the SUT for this example, what is going on with the service is important as well. If information from the server side (from a log or custom instrumentation) is available synchronously (or even *nearly* synchronously, with a short wait on the information) then those details can go into the artifact as well as a separate, strongly typed unit of information (e.g., in an XML element, as is done with the sample project).

The resulting artifact has sufficient information to make root cause apparent, with no debugging sessions needed. The techniques outlined here scale to more complex tests, to enable actionable results and application of the four following patterns of MetaAutomation.

3 Conclusion

Atomic Check is the first of the five patterns of the MetaAutomation pattern language.[14]

Applying this pattern, or adapting current practices and infrastructure to the pattern, will bring greater effectiveness to the quality team and value for the software product. Automation will run faster, scale better, and give decision-makers for quality what they need to know.

Looking forward to the following four patterns of the language – User Pool, Parallel Run, Smart Retry, and Automated Triage – offer potential paths to further acceleration of communication around the team, greater perspectives and broader horizons on current automation practices around software quality measurement and regression testing.

[14] See the book MetaAutomation by Matt Griscom, to be published in 2014, for more information

References

Books

Alexander et al., *A Pattern Language*, 1977

Binder, Robert V., *Testing Object-Oriented Systems: Models, Patterns and Tools*, 2000

Gamma et al, *Design Patterns*, 1977

Garg & Sharapov, *Techniques for Optimizing Applications - High Performance Computing,* Prentice-Hall 2002

Graham, Dorothy and Fewster, Mark, *Experiences of Test Automation: Case Studies of Software Test Automation*, 2012

McCaffrey, James, *Software Testing: Fundamental Principles and Essential Knowledge*, 2009

Meszaros, Gerard, *xUnit Test Patterns*, 2007

Myers, Glenford J., *The Art of Software Testing*, 1979

Page, Johnston, and Rollison, *How We Test Software At Microsoft*, 2009

Riley & Goucher, *Beautiful Testing*, 2010

Whittaker, James, *How Google Tests Software*, 2012

Kaner, Cem, *Exploratory Testing*, Florida Institute of Technology, Quality Assurance Institute Worldwide Annual Software Testing Conference, Orlando FL, November 2006

Periodicals

International Software Testing Qualifications Board, Standard glossary of terms used in Software Testing: Version 2.2, 2012

Blog Posts

James Bach and Michael Bolton, “Testing and Checking Refined”
http://www.satisfice.com/blog/archives/856