**Transforming a Test Automation Maintenance Nightmare into Success**

[article]

By [**Vinay Shah**](https://www.stickyminds.com/users/vinay-shah) - January 13, 2020

[**Share URL**](https://well.tc/5gs7)

**Summary:**

Best practices for test automation emphasize reliability, portability, reusability, readability, maintainability, and more. But how can your existing automated test suite adopt these qualities? Should you address these issues with your current tests, or create an entirely new set of tests? Here are some questions that will help you determine if your test automation maintenance program is operating as it should be.

“Automation” is not a new buzzword in the industry. With the evolution of e-commerce and rapid access to mobile technology, delivering software applications as quickly as possible has been a trend for some time. But it’s difficult to appreciate the solution without truly understanding the problem. One size doesn't fit all, and there is not one perfect "best practice" solution that applies to all automation problems. We must weigh the cost, effort, and risk against potential benefits.

There are tons of online resources about best practices for test automation that emphasize reliability, portability, reusability, readability, maintainability, and more. When I first started creating automated tests, I found this information helpful as well as stressful. How could it be practical to adopt all these practices for your tests from the get-go? If you are a test automation engineer, I’m sure you have faced some of these challenges as well at some point in your career.

Let me start with my journey of writing browser automation tests, then get into what I learned from my mistakes and how I overcame challenges.

Writing tests was initially time-consuming, and I was always trying to improve as I cycled through them during maintenance. Just like any other development task, creating tests also has deadlines and management expectations, and balancing these factors is crucial for success in a test automation project.

In order for my first project to meet the schedule, I rushed to create the tests and didn't consider some of the best practices mentioned earlier. My tests were stable and passed 100% of the time—until the application under test (AUT) started changing a few months later. Now, the real quality of my tests came to the surface, and they became a maintenance nightmare.

Whenever a test failed, we spent lots of time trying to understand the cause of the failures so we could determine whether it was due to regression, an expected change in the AUT, or environmental issues such as a new browser or system updates. After weeks of troubleshooting and frustration, we spent some time to identify the issues that manifested from our tests.

Here is what we discovered:

* Most of our tests were too big, consisted of too many steps, and tried to validate too many different functionalities. Many of these steps also depended on the successful execution of previous steps, and there were many inefficient wait conditions added, such as static delays, which made execution unnecessarily long.
* From a test report standpoint, there were many times it was hard to understand the test failures. We weren't able to quickly determine what the cause of failures were without spending a lot of time on the report. Many test steps had generic names or descriptions based on element locator—for example, clickButton - //div[2]/button—and when such a test failed, it wasn't clear which button on the page it was referencing.
* These tests weren't very portable. If someone wanted to execute tests from their workstation, they had to set up many additional environmental data locally because there were preconditions that needed to be set up as part of the build process outside the test. Another issue was that tests had many hardcoded references, such as resource name, that made it inconvenient for anyone to run these tests outside their specific test environment.
* Tests lacked execution repeatability. Rerunning the test in the same environment was challenging and many times didn't work because people were not resetting the environment after its initial execution. This prevented quickly rerunning tests after updating them or after a new AUT build without manually setting all preconditions outside the test.

Once we identified the issues, next we had to decide how to tackle them. Our choice was either to address those issues with our existing tests or to create an entirely new set of tests.

Since we had a large number of complex tests, we had to consider the time it would take to recreate them and assure management that the new tests would not have maintenance issues moving forward. After assessing effort vs. risk factors with the team, we unanimously decided that recreating all tests would not be a viable option. We didn’t want to accidentally miss any existing functionality covered by these tests.

That left us with the option of **fixing existing tests using best practices** that would address most of our maintenance challenges. Here is how we mitigated our identified problems.

First we had to **break down our existing poorly crafted tests into smaller modules** that could independently run and test specific functionality. Note that this may not be feasible for every organization, depending on the volume of tests and the amount of time it would take to refactor them. In such cases, **the best thing to do is to leave old, stable tests as is and only refactor those that need immediate attention**. Sometimes recreating tests may not be a bad option, considering the amount of time it would take to fix all existing tests.

Then we separated out **any commonly used test code and procedures into their own modules**. Instead of copying and pasting these pieces of code all over, we simply started referencing it.

Then we created **setup** (precondition) and **teardown** (resetting any changes the test introduced in AUT) conditions as part of the test. Overall, this makes the test execution a bit longer, but it’s worth it to have standalone tests that can quickly execute without any manual steps or running lengthy build process.

We **removed hardcoded references** (host name, port, http, https, etc.), parameterized them, and made them configurable.

Finally, we **optimized the test execution time and remediated fixed delays** by applying proper smart wait conditions, such as having to wait until an element is visible, clickable, etc. This helped us to eliminate unnecessary long delays during test execution.

To **make tests consistent, we defined a guideline and review process** for everyone who is going to maintain these tests or create new tests in our infrastructure. The guidelines included test and element naming, element locator technique (use XPath vs. CSS), and documentation and comments. This consistency exercise made it easy for anyone on the team to read and maintain our tests.

In addition, **we set a team goal for maintaining 100% passing tests, addressing any failing test as quickly as possible**. A test could be failing or unstable due to environmental issues, poor test or wait conditions, a constantly changing AUT, etc., so it’s always best to identify such failures in a timely manner and attempt to address them immediately. This will retain the team's good faith in your automated tests.

These are the issues and solutions for our **suite of automated tests**, but again, there is no one correct formula that will work for everyone. How do you know if you have adopted a best-practice approach with your set of automated tests?

Asking these questions will help you determine if your test automation maintenance program is operating as it should be:

* Are the tests easily maintainable by your team?
* Are they stable and repeatable?
* **Are there any intermittent failures without any changes to the AUT**?
* Can others run these tests in their environment without performing a bunch of setup work? (With browser tests, as long as the AUT is set up, anyone should be able run these tests from any machine with the supported browser)
* Do tests take a long time to execute?
* Can someone understand a possible failure based on the test report?
* Do you have to apply the same fix in multiple places? Can you refactor the common piece of code in one module and reference it?

Don’t be afraid to ask for feedback from developers and other team members about your tests. There is always room for improvement, and as a test automation engineer, you should always strive for it.

**Continuous-Testing**

**Continuous testing involves testing a software application beginning in its early stages and automating testing throughout the development lifecycle**. This helps the team examine the quality of the product at every stage of the continuous delivery process. And this process is not limited to only testers and developers; it also involves the contribution of stakeholders, operations, and even the customer. Continuous testing is an integral factor in the DevOps equation.

If continuous testing is performed properly, it delivers quick and uninterrupted insight into the quality of every new build of your software. This information can help you analyze whether the application is prepared to go through the delivery pipeline.

For example**,** when the code in a source code server like Jenkins is verified by developers, a set of automated unit tests is executed in the continuous process. If the tests don’t pass, **the build will be rejected**, and the developers will be notified about it. If the tests pass, the code will be sent to the QA servers for functional and load testing. Then those tests are executed in parallel, and if they pass the build, the application will be deployed in production.

A change to a dependency often also runs all the downstream builds too. If a change that a developer makes breaks something downstream, it isn't easily seen when modifying a library (say changing an underlying datatype from a SortedSet to a HashSet (only providing the contract of Set) and someone downstream worked on the mistaken assumption that the Set was sorted). Not running the (downstream) tests on the CI server would let that bug fester for awhile.

Unit tests are tests, which are **separated from any IO operations** (File system, Web service or database). Because unit tests are tests which developer need to run every 1 minute and execution time of tests need to be very short (seconds). While tests with involved IO operation takes longer time.

Tests which depend on IO operations have place to be, but they called "Integration tests".

In order to do that, you may use mocking. Mocking is the technique where you replace one part of a system by something which only provides fictive results. In your normalization tests, you may have a mock which provides labeled data, another mock which provides data which is labeled, but incorrectly, and another mock which doesn't label anything.

Those three mocks will serve you to unit test your normalization independently of the actual labeling process, without even carrying if it is already implemented or not, nor if it is working as expected.

This way, you can unit test the part which saves data and then develop it, then start to unit test labeling and implement it, and finally unit test normalization and write the corresponding code.

Once done, you will then use your actual integration tests to see if the whole system is working.

Make sure developers understand that breaking the build (or failing regression tests) is not OK, and that as soon as it happens, their top priority is to fix it. It doesn't matter whether they are working on a high-priority feature that their boss asked to ship for tomorrow: they failed the build, they should fix it.\

Thanks for the thoughts Ewan. But when everything is tested on master, that's a lot of stuff merged already. And depending on the workload **and length of the sprint (which in our case is 2 weeks)** it can be a lot to test right? Also as soon as something's merged, the feature task would be marked done. So we lose track of all the important information in the task as well. Because that doesn't stay as a source of truth for QA anymore once its marked done.

What you could do is make sure your feature branches stay up-to-date with master by rebasing them every time master is updated.

Yes, it creates more work, but it would allow you to keep your testing in a feature branch, while ensuring that you're testing the code **as it would be in production**. And it would ensure that the merge to master after the QA pass remains easy.

The only thing left is to decide whether it is the Dev or QA's responsibility to rebase the feature branches. I would suggest the Dev since they would know best how to take on changes from master without affecting the new feature.

In my experiences, a good ratio is about 3-5 developers to 1 manual tester. This seems to work out well as long as the tester is involved from early in requirements development to begin to start identifying test strategies, approaches, and cases. The work to develop the tests begins at the same time as development on the functionality. In cases where the development is done and the work is handed off, I've found that the developers need to shift away to support testing as the tester has questions or isn't sure about behavior in edge cases unless it's explicitly documented somewhere.

Second, the testing in feature branches before merging to master is concerning. This is increased since there are conflicts that need to be resolved when merging the feature branch into master.

It seems like the code that is being tested is not the code that ends up in the mainline of the product. This means that you can't be sure that your tests are finding the problems that exist in the post-merged product. **There are two ways to correct this. First, regularly merge the master branch into long-running feature branches and test. You may not need to retest everything, but if there are conflicts, you can analyze the diffs and choose what tests must be re-executed.** Second, if you are working on a planned release cycle, something like the gitflow model may be useful in ensuring that master reflects releases while having feature and development branches that are synchronized with each other.

Third, automate.

Work on automating your test suite. Make it very easy to run various test suites. A good focus would be on **regression** - finding and ensuring that previously fixed issues don't return. But automate as much as possible from the unit through the system level and make it easy to run. **Integrating into a build process** would also help to ensure that each branch is in a good state.

* decrease the test case execution time by 60%,

✅ decrease the number of unnecessary Selenium commands by 900%,

✅ increase test reliability to 100%

**But how can you break up your large end-to-end tests:**

Here is a simple scenario:

1. Open Amazon.com
2. Assert that the page opens
3. Search for an item
4. Assert that item is found
5. Add item to cart
6. Assert that item is added
7. Checkout
8. Assert that checkout is complete
9. The first problem is that many automation engineers assume that you must do an entire end-to-end flow for this automated test.
10. You must complete step 1 before step 2 and so on… Because how can you get to the checkout process without having an item in the cart?
11. **The automated testing best practices approach is to be able to inject data to populate the state of the application before any UI interactions.**

#### How to manipulate test data for UI automation?

You can inject data via several options:

1. Using something like a RESTful API to set the application into a specific state
2. Using JavaScript
3. Injecting data into the DB to set the application in a certain state
4. Using cookies

If you can inject data between the seams of the application, then you can isolate each step and test it on its own

**Some Options:**

1. Use an API to send a web request that will generate a user
2. Use an API that will generate an item in your Amazon cart
3. Now you can pull up the UI to the cart page and checkout using web automation
4. Clean up all test data after

**This is the best practice approach. You tested the checkout process without using the UI for steps one through three**.

#### How To Coordinate API and UI Interactions In One Test?

What if you wanted to coordinate API and UI actions in a single test? Well, that would look something like this.

The most common situation is not to alternate Selenium and API calls. Normally we do API stuff to set up state. UI stuff to assert functionality. API stuff to clean up. It’s not common to do Selenium, API, Selenium, API, Selenium and so on.

However, it’s possible for a user case for sure!

Just remember that Selenium or any other UI framework has nothing to do with your App API. A UI automation library is all about the front-end, the UI. The API part of your test is all about managing state and test data, the back end.

#### A Good Page Object Pattern:

1. The page object class has a great name that tells us exactly what the HTML page or HTML component does

* If you cannot name your page object so that it’s 100% clear what’s inside of that page object, then it’s likely your page object does too much.

1. The page class contains methods to interact with the HTML page or component

* The only public methods that are allowed in your page objects are those that an end-user can perform to your web application.
* On this HTML page, the only two actions that an end-user can perform are **Open()** and **Login()**
* The user cannot **ConnectToSQL(),** **OpenExcel()**, **ReadPDF().**Hence, such actions should never be found in your automated UI tests.

1. The page class contains properties to interact with the HTML page or component

* Page objects should allow us the capability to interact with the application through a single interface, the page object. This means that if we want to interact with the login page, we interact with it through SauceDemoLoginPage class.
* For example, we might want to login SauceDemoLoginPage.Login("username", "password");
* What we want to avoid happening is for all the locators to live in a separate class for example. This makes the code more complicated unnecessarily. Although there are many articles claiming that separating methods from properties for a page object helps to maintain Single Responsibility Principle, that’s actually not correct. The SRP is viewed from the actor who can cause a change to our code. Technically, there is only a single actor that can ever break our class, the Developer. There is no other actor that will solely modify HTML elements without modifying the HTML page.
* In conclusion, avoid separating out locators from methods for a page object. It’s over-optimization. However, a page may be composed of multiple components that are relevant to that page object.

1. Properties and methods live in a single class
2. The page object exposes only methods that an end-user would use to interact with the HTML
3. A page object doesn’t need to be an entire HTML page, it can be a small component as well

The benefit of using Page Objects is that they abstract implementation logic from the tests. The tests can be focused on the scenarios and not implementation. The idea is that the scenario doesn’t change, but the implementation does.

#### Who is sorting through all of these failures?

Is there someone on the team that is sitting and sorting through all of these failures?

**~104,000 non-passing tests…**

**So what is the reason that these tests failed?**

**A**. Because there is one bug in the application that caused all of these failures?

**B**. Because two or more bugs are causing all of these problems?

**C**. Because there are **~zero bugs** found and the majority of the failures are a result of worthless automation efforts?

I’d bet $104,000 that it’s **option C** 😂

**It gets worse:**

How many automation engineers do you need to sort through 104,000 non-passing tests in one week?

When I ran a team of four automation engineers, we could barely keep up with a few non-passing automated tests per week.

So let’s be honest… nobody is analyzing these non-passing automated tests…

**If your automation is not providing a correct result more than 99.5% of the time, then stop automating and fix your reliability! You’re only allowed 5 false positives out of 1000 test executions. That’s called quality automation.**