World Academy of Science, Engineering and Technology
International Journal of Computer and Systems Engineering
Vol:18, No:2, 2024

Factors Affecting Test Automation Stability and Their Solutions

Nagmani Lnu

Abstract**—**Test automation is a vital requirement of any organization to release products faster to their customers. In most cases, an organization has an approach to developing automation but struggles to maintain it. It results in an increased number of flaky tests, reducing return on investments and stakeholders' confidence. Challenges grow in multiple folds when automation is for User Interface (UI) behaviors. This paper describes the approaches taken to identify the root cause of automation instability in an extensive payments application and the best practices to address that using processes, tools, and technologies, resulting in a 75% reduction of effort.

Keywords**—**Automation stability, test stability, flaky test, test quality, test automation quality.

I. Introduction

APPLICATION undertest is a large, complex, and monolithic application to accept payments. It has 963 functional test cases which on paper were automated with Selenium, however, they never worked. Because of instability, Quality Assurance (QA) engineers were executing these test cases manually and ended up taking three days to complete one cycle of regression.

Unstable automation is a classic example of an automation challenge that most companies are facing. All companies are investing a lot in test automation. Based on market estimates, software companies worldwide invested $931 million in automated software testing tools in 1999, with an estimate of at least $2.6 billion in 2004 [4]. Testing is costly, and though its estimate varies from project to project, statistically speaking, testing occupies 20% of the overall development time for a single-component application, 20% to 30% for a two-component application, and 30% to 35% for an application with GUI [2]. The number can be as high as 35% to 50% for a distributed application with GUI [1], [2]. Automating test cases ensures a faster execution and hence a reduction in the overall effort. Nowadays, most companies follow an Agile development approach in which applications are getting developed in multiple iterations. As per the statistics published in 2022, at least 71% of US companies are following Agile now [3]. In general, automating previous iteration functionalities test cases or automating test cases within the sprint ensures a speedy execution and, eventually, enables faster delivery.

Automation instability and Flaky tests are big problem across industries; even big giants like Google and Microsoft are not untouched. A survey published in October 2021 shows that 41% of Google and 26% of Microsoft test were Flaky [6].

Tests can be flaky because of *Flaky Test environment* and *Unstable tests* [5]. While individual test engineers cannot control the test environment, there are better practices that can effectively reduce inconsistent tests.

II. Root Cause Analysis and Solution

The project team was tasked to stabilize the automation that mainly had two milestones: Root Cause Analysis and Develop a Robust Solution to Ensure Reusability. The team adopted a three-step approach: a) Execute, b) Analyze, and c) Document to reach the first milestone which was finding the different root cause of the failures. Under the step *execute*, the team decided to run all the automation at least once a day to gather the execution results and *analyze* the failure root cause as step 2 and finally *document* those as step 3 so they can further group the related causes for future action as part of the second significant milestone which is developing a robust solution to ensure reusability. These steps for Root Cause Analysis are shown in Fig. 1.

Fig. 1 Three Steps process for the Root Cause Analysis

After carefully studying many automation failures, we observed that we failed to maintain the automation for many reasons. The following are the most important ones:

- *Lack of proper repository and code management principle:* QA engineers were developing automation scripts and keeping them running locally. We have multiple team members doing the automation, but they were not frequently checking their code to a single repository, hence creating a risk of breaking the other team member's automation. This matches a survey report published in October 2021 that says that out of the 96 sampled order-dependent tests, 94 reportedly caused a test failure in the absence of a bug, i.e., a false alarm [6]. It is a huge problem, and an organization should handle this by creating a code management policy. We established this practice by

Nagmani Lnu is Director of Quality Engineer at a FinTech Company, San Antonio, USA (e-mail: nagmanijobs@gmail.com).

Open Science Index, Computer and Systems Engineering Vol:18, No:2, 2024 publications.waset.org/10013510.pdf

creating a code repo in Azure DevOps to maintain the automation code sanity immediately.

- *Traceability between Manual and Automated Tests:* The application was going through a phase of development and maintenance. As per the world quality report published in 2018, 61% of companies state that they were unable to maintain automation because of frequent changes [12]. In standard practice, test engineers create test cases and automate them. Every time the application changes, test engineers are supposed to update both manual and automated test cases. While it was easy to locate and correct manual test cases, engineers often needed to pay more attention to update the corresponding automated test cases because that requires traceability between the automated and manual test cases were missing. Tools like *Azure DevOps* have features of associating automated test cases with manual test cases directly from the code editor like Visual Studio [8]. Azure DevOps is a popular tool with a market share of 30% [10] that provides services like Board, Repo, Pipeline, Test Plan and Artifacts with one tool [11]. Fig. 2 is taken as an example from the Azure DevOps documentation [8] to show how an automated test can get linked with the manual test cases.

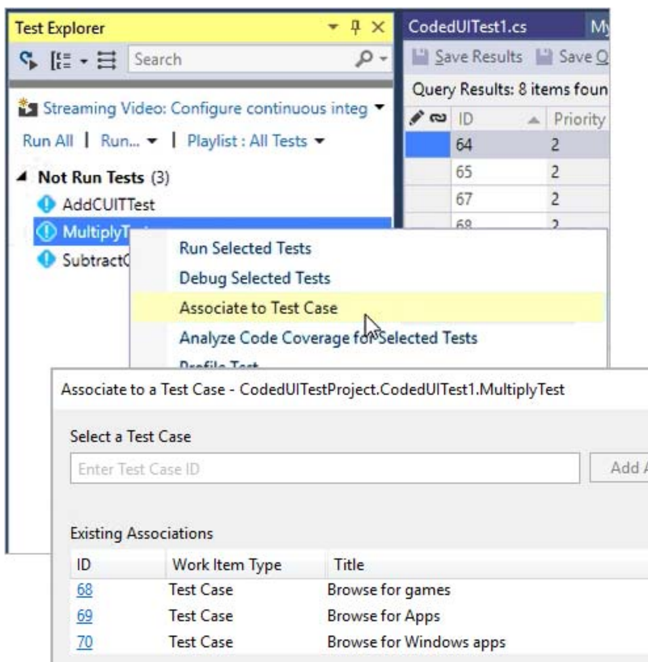

Fig. 2 Associate Automated test script with manual Test Cases in Azure DevOps

Using the feature of associating automation and manual test cases has enabled us to identify the application changes immediately while updating the manual test cases; it has increased the automation reusability to almost 100%.

- *Infrequent Test Execution:* In an ideal situation, test scripts should be a part of the build and deployment pipeline. However, because of the longer execution cycle, the team tried to manage it separately. As a result, test cases were executed infrequently and often failed when trying to run. To fix this, we created a stand-alone test execution pipeline in Azure DevOps to trigger the automation automatically. We captured some vital metrics like *Failing Since* or *consecutive failure days count.* Failing since was a critical data point to understand the magnitude of instability. Sometimes, test cases fail because of other factors, like test environment slowness, wrong build deployed in the QA environment, or vendor environment problems. One should not jump to conclusions immediately and should spend time analyzing the root cause. We executed all the test cases for the first week just to gather the test results for further analysis. We observed that at least 10% of failed test cases in the previous run were getting passed in the following execution. However, the overall pass percentage varied between 15% and 25% during this time. That means most of the test cases were failing.

After carefully analyzing the data, we were able to find the exact cause of inconsistency as shown in Fig. 3.

Test Scripts	Failing Since	Manua Test Case ID	Root Cause
Check Balance	10th Dec 2022	1122	Data Changed
Update Payment method	9th Dec 2022	3211	Iframe locator changed
Submit transaction	10th Jan 2023	3322	Real defect

Fig. 3 Template to capture Failing Since and Root Cause

Additionally, with Azure DevOps, we got many other inbuilt metrics like *Overall Pass rate, pass rate of each test, Test case execution trend, failure logs*, etc. When we run the tests from their designated pipeline, our analysis was made very easy to establish subsequent actions for stabilizing the tests [7]. Fig. 4 is one such report taken from the Azure DevOps documentation [7].

Test	Failed	Pass rate ↑	Total count	Average duratic
CIWorkflow.Tfs.WebPlatform.L2.Tests.dll	383	91.5%	15020	141.93s
ValidateRetentionTabExperienceForTfvcProject	368	75.49%	1502	42.35s
ValidateOptionsTabExperience	15	99%	1502	62.91s
Graph.Vssf.Sdk.L2.Tests.dll	87	99.03%	9012	40.8s
Graph.Tfs.Client.L2.Tests.dll	9	99.7%	3004	28.27s
WorkItemTracking.Tfs.ExtendedClient.L2.Tests.dll	234	99.82%	135686	197.56s

Fig. 4 Pipeline Test Execution report

- *Dynamic Data in the Data Table:* As per the World Quality report published in 2018, 48% of companies are facing test data challenges while doing the automation, in 2016 it was 40% [12]. As per the analysis with this project, this was the number one cause of unstable tests. Functional tests depend on test data, and dynamic data change over time. The basic principle of automation is repeatability; we cannot achieve repeatability with dynamic data, and thus engineers need to handle it very carefully.

Test Data Management (TDM) is a growing concept addressing this situation that creates, manages, and delivers test data to application teams. We could not implement TDM in a very short amount of time. After evaluating a series of failed

Open Science Index, Computer and Systems Engineering Vol:18, No:2, 2024 publications.waset.org/10013510.pdf

Open Science Index, Computer and Systems Engineering Vol:18, No:2, 2024 publications.waset.org/10013510.pdf

tests during this exercise, we found the following challenges with respect to the test data:

- *Shared test data:* In most financial companies, certain test data will be shared among the team (e.g., portal login). This is not a good practice; however, it is the reality. Once shared, control on the test data reduces drastically.
- *Unable to create new test data:* Financial companies often integrate their application with a vendor application, creating complexities as the vendor gives very limited test data that they process from their end. Also, we could not mimic this type of data during system testing. An example is a test credit card, or test account numbers.
- *Test data cannot be reused:* Some test data cannot be reused unless it gets reset back to its original state. For example, one will not be able to reuse an account number if it gets closed, or an application will block a duplicate payment on certain loan accounts.
- *Hard coded test data in the script:* QA Engineers had hard coded certain data like dates in the script, resulting a failure in the second run. Also, some validation like person name, account number, and status were so tightly linked with the parent test data, resulting in a false alarm if the parent test data gets corrupted and then, the QE needs to replace it.

We implemented different solutions for each of the problems stated above. To solve hard coding, we enforced coding practices that stopped QE engineers from hard coding the value. Instead, they need to create a function that can make those dynamic test data on the fly, for example, a date function to return desired date based on the business rules or connect to the database in the backend to check the properties related to parent test data and pass it in the script instead of hard coding.

As mentioned above, we could not reuse some of the test data. This was the trickiest problem, and we had to implement a unique solution to each case. For example, we called API to generate account data on the fly and pass it to UI for further processing, but in some scenarios, we will not have API available to generate the test data; for that, we created a setup script to reset the data in the database before executing the main test case.

The last but most widespread problem related to test data was that most of our test data were shared. Business flows are dynamic and differ from person to person based on their credentials or privileges. For example, the application will appear differently to an admin user than to a regular user. In our application, specific business flows or transactions will only be allowed if the users are qualified. Before this exercise, QA had acquired the test data and used those to develop automation scripts. The team had assumed that those test data were allocated only for automation. However, those test data slowly went public, and each team started modifying those based on their needs. QA engineers in our application were unaware of these facts, reporting those failures as defects that eventually turned into false alarms. It became a significant cause of dissatisfaction as automation was supposed to save time, but in turn, even the developer had to spend their time debugging false alarms.

To fix this issue, we created utilities to dynamically generate/reset test data. The decision to develop the utilities was very innovative as, most of the time, QA engineers think traditionally and use a Setup method that comes with the framework. However, the limitation of the Setup method is that it gets tied very closely to the test cases. Some test data, like portal user or their respective credentials, were identical to multiple test cases. So, once reset, it will be suitable for all the related test cases. Also, adding setup scripts increases test execution time, so one should avoid it unless needed. After analyzing all the failures, we created 12 utilities for 963 test cases. The team created a separate pipeline to run these utilities before each execution to reset test data to their original condition, as shown in Fig. 5 of the overall Test Execution steps.

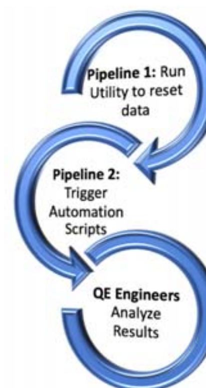

Fig. 5 Overall Test Execution Steps

It has a drastic impact on the overall test stability. By doing this alone, the pass rate for test cases was increased by more than 60% and freed up almost 100% of the developer's time spent earlier on debugging the false alarms.

- *Dynamic Element Identifier and Other Minor Improvements:* This was minor; however, automation stability highly depends on the locator selection strategy. We switched to advance Xpath instead of standard Xpath to resolve this issue [9]. Other than this, we also improved coding standards to capture different application errors in a more readable fashion to troubleshoot any failures more effectively.

III. Conclusion and Results

The whole exercise took approximately 120 hours. Once everything mentioned above was implemented, we started seeing reduction in the overall timeframe of test execution, which used to be three days in the past, but is now reduced to six hours, resulting in 24 hours being saved. The project is making at least one release every week and has so far saved around 1000 hours overall.

References

[1] B. Shea, "Software testing gets new respect," *InformationWeek*, July 2000.
[2] "Time Estimation for software testing," devmio - Software Know-How, 08-Feb-2016. (Online). Available: https://devm.io/testing/time-estimation-for-software-testing-128078#:~:text=Statistically

Open Science Index, Computer and Systems Engineering Vol:18, No:2, 2024 publications.waset.org/10013510.pdf

%20speaking%2C%20testing%20occupies%2020,as%2035%20to%205 0%20percent (Accessed: 18-Jan-2023).

[3] Zippia. "16 Amazing Agile Statistics (2023): What Companies Use Agile Methodology" Zippia.com. Nov. 27, 2022, https://www.zippia.com/advice/agile-statistics/

[4] R. Krish, “Test automation framework – challenges in the ever changing technology scenario,” Test Automation Framework – Challenges In The Ever Changing Technology Scenario - page 4. (Online). Available: https://www.siliconindia.com/guestcontributor/guestarticle/383/test-automation-framework-%E2%80%93-challenges-in-the-ever-changing-technology-scenario.html (Accessed: 26-Jan-2023).

[5] “How to find flaky selenium test suite,” LambdaTest, 06-Sep-2022. (Online). Available: https://www.lambdatest.com/blog/flaky-selenium-test-suite/ (Accessed: 26-Jan-2023).

[6] Owain Parry, Gregory Kapfhammer, Michael Hilton, and Phil McMinn, “A survey of flaky tests a survey of flaky tests,” A Survey of Flaky Tests. (Online). Available: https://dl.acm.org/doi/fullHtml/10.1145/3476105 (Accessed: 26-Jan-2023).

[7] Vinodjo, “Test analytics - azure pipelines,” Azure Pipelines | Microsoft Learn. (Online). Available: https://learn.microsoft.com/en-us/azure/devops/pipelines/test/test-analytics?view=azure-devops (Accessed: 26-Jan-2023).

[8] steved0x, “Associate automated tests with test cases azure test plans,” Azure Test Plans | Microsoft Learn. (Online). Available: https://learn.microsoft.com/en-us/azure/devops/test/associate-automated-test-with-test-case?view=azure-devops (Accessed: 26-Jan-2023).

[9] K. Rungta, “Xpath in selenium,” Guru99, 21-Dec-2022. (Online). Available: https://www.guru99.com/xpath-selenium.html (Accessed: 26-Jan-2023).

[10] M. Fayaz, “Azure devops vs AWS Devops,” Cloud Training Program. (Online). Available: https://k21academy.com/amazon-web-services/aws-devops-vs-azure-devops/ (Accessed: 26-Jan-2023).

[11] Chcomley, “What is azure devops? - azure DevOps,” Azure DevOps | Microsoft Learn. (Online). Available: https://learn.microsoft.com/en-us/azure/devops/user-guide/what-is-azure-devops?view=azure-devops (Accessed: 26-Jan-2023)

[12] Qentelli, “It Is Automation, Not Automagic: Avoiding Failures in Test Automation Projects,” https://www.qentelli.com/thought-leadership/insights/it-automation-not-automagic-avoiding-test-automation-failures (Accessed: 23-Dec-2023)