jamo solutions

How to reduce the cost of mobile test creation and automation

Introduction

Automation of user acceptance testing can be a reliable way to reduce the cost of testing, increase test coverage and effectiveness, and shorten test cycles. Many development organizations consider automation a vital step in establishing a mature QA practice, and it certainly can bring great benefits when it is effectively leveraged.

This paper focusses mainly on the different aspects of creating and maintaining of automated test cases. It is not intended to cover such topics as the definition of test cases, the process of determining which test cases make good candidates for automation, or the analysis and qualification of test execution reports. In the first section, we will recount how traditional automation technologies influence the process of test automation, followed by the introduction of new automation methods that help remedy some key roadblocks that are not addressed by standard automation practices.

Traditional test automation technologies

Without a doubt, test automation tools save time, increase test coverage, and ultimately help deliver better quality applications faster. They do however tend to be quite costly, requiring sizable upfront investments, which typically take between two and four years to fully pay off. The way most traditional test automation tools work is by giving the QA engineer access to the application layer that is used by developers to define the app's user interface. The QA engineer must possess strong technical knowledge to be able to work with this layer to determine which objects need to be verified, which data values must be used, and which user actions trigger changes in specific objects.

Naturally, each software platform uses different technologies and protocols to define the app UI layer, and if the app is intended to work across multiple platforms, such as Android, iOS, or the Web, it is usually not possible to reuse the same automated script, requiring QA engineers to re-create the same script on different platforms.

Another issue with using the app UI layer for test automation is that developers' object definitions inside the UI layer often don't directly match the UI objects that are seen by the end user. Developers build UI layers to get the system to display requested information in the fastest and most efficient way, but it can cause issues when trying to use these objects for testing. Consider the following examples:

1. The main menu in iOS, Android, or Web apps is always stored in memory, however it is not displayed until called by the user. When QA engineers use the UI layer for testing, the menu is visible to them at all times, and testers need to write automation scripts to work around this matter.
2. Most lists on Android and iOS contain only visible cells. For memory efficiency reasons, other elements are not mapped into the UI memory. If a test needs to access an element that is not currently visible, a QA engineer will have to make special provisions in the code to accommodate this issue.
3. Users expect an attractive and eye-catching app interface, which causes the hierarchy of UI elements to become more complex, forcing testers to scrutinize multiple objects to find the ones needed for test automation.

The mapping of the technical layer to the visual objects is further complicated by the following:

- Responsive design: the app might display information slightly differently on smartphone screens that are different sizes.
- Dynamic data: the data that is displayed in the app is updated either because it is delivered in real time, or has changed as a result of the user's actions. Dynamic data makes is difficult for a QA engineer to use labels, text, or list cell positions to identify correct UI objects, further complicating the process of automated test creation.

How much does it cost to create test scripts using traditional automation technologies?
Let's try to estimate the total cost of test automation with the tools that are most widely used by today's testers. Inherently, the total cost of test automation depends on the number of test cases, but for this scenario, let's assume that a total of 60 test cases have been classified as good candidates for automation. Upon initial review, the QA engineer identifies a number of test cases that are similar enough to be covered by the same automation script. If some test cases use similar windows, common scripts can be created for specified windows access. Taking into account these similarities, a typical compression factor works out to be around 75%-80%, meaning that if the total number of test scripts that have been qualified for automation is 60, the QA engineer will need to create between 45 and 48 automated scripts to cover all defined test cases.

Experience and industry data suggest that an average time to create one automated test case is six hours. If a test case has to deal with responsive design or dynamic data, then the average time to create an automated test case can double to 12 hours. The table below summarizes the costs of implementing automated test cases.

	Best-case scenario	**Worst-case scenario**
Number of test cases	60	60
Compression factor	75%	80%
Number of automated test cases	45	48
Hours required to implement one test case	6	12
Total hours needed for automation	270	576
Total days needed for automation	34	72

In this scenario, it will take between 34 and 72 person-days to automate 60 test cases, and that's just the cost of initial automation. Since the real ROI comes from test execution, let's look into the added cost of test maintenance between app releases and test cycles.

Even within a single release cycle, tests need to run multiple times, often causing automated scripts to break. Here's a look at the three most common reasons why this might happen:

1. The test script was written in a way that doesn't take into account the fact that certain data values cause an additional screen or dialog box to open.
2. Developers made changes to the UI layer without disturbing the display of the UI object, but affecting the way the QA engineer identified the same UI object in the test script.
3. The test script is executed on a different version of the mobile OS than the one it was originally created for, causing the differences in the technical UI layer to keep the script from running correctly.

In our experience, even within the same release, between 10% and 20% of all test cases may break between execution cycles, and will require maintenance. Now, let's examine the scenario where a new major version of the app is being released. Automated test cases need to be run against a new version for regression testing – making sure the app still works despite the functionality changes made in the new release.

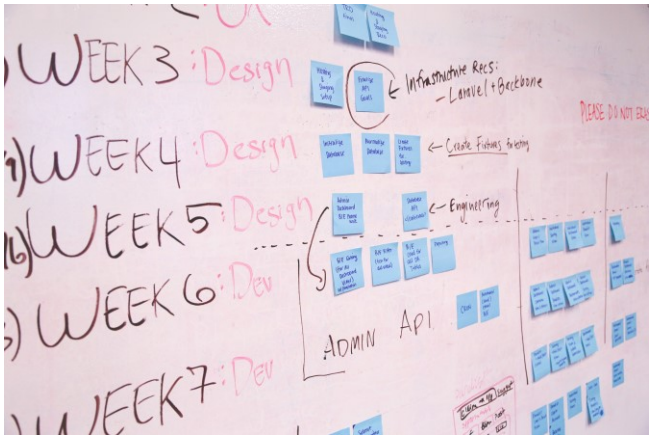
Examples of the common factors that cause automated tests to break between different app versions include the following:

1. New functionality introduces changes to the UI, requiring test scripts to be modified.
2. The visible UI might be identical, but the developer has made changes to the technical UI layer.
3. A particular screen still performs the same function, but the design elements of the UI have changed, causing changes in the technical UI layer, on which the script has been built.

Between different major versions of the app under test, as much as 65% to 80% of all test cases may need to be updated or rewritten. The table below illustrates the test case maintenance costs between test cycles and app releases.

	Best-case scenario	**Worst-case scenario**
Number of test cases	60	60
Total days needed for automation	34	72
Percent of rework between test cycles	10%	20%
Number of days required for rework and maintenance between test cycles	3	14
Percent of rework between app releases	65%	80%
Number of days required for rework and maintenance between releases	22	58

The growing popularity of DevOps, agile development, and continuous delivery practices are putting traditional test automation tools under increased pressure. Spending 22 days maintaining scripts for each release is not only excessively costly; it can jeopardize the main objective of modern application delivery models that are based on rapid release and feedback cycles.



The new generation of automation tools improves testing time and agility

Jamo Automator works differently than traditional test automation solutions. To identify UI objects, the tool doesn't rely on the details of the developers' UI definition, making scripts more resilient to changes in the technical UI layer.

Jamo Automator applies an advanced algorithm built using embedded AI to recognize correct UI elements during test replay. The tester doesn't need to know a lot about the technical definition of the UI layer created by the developer. Cosmetic UI changes, dynamic data, responsive design, and smartphone screen resolutions no longer pose threats to the test execution. As long as the app under test offers the same user experience across platforms, the same script will work without modification on iOS, Android, and other supported environments.

The user-friendly experience of Jamo Automator is based on building automated scripts by using the graphical flow of test steps. With each step represented by a corresponding screen, a tester can identify a relevant UI object by simply clicking on the screen, no programming required.

This new-generation automation tool reduces the time required to automate each test case to just $1\text{-}2^{1/2}$ hours. Consider the time savings afforded by Jamo Automator as outlined in the table below:

	Best-case scenario	**Worst-case scenario**
Number of test cases	60	60
Compression factor	80%	85%
Number of automated test cases	48	51
Hours required to implement one test case	1	2.5
Total hours needed for automation	48	127.5
Total days needed for automation	6	16

Automated test cases built with Jamo Automator can be executed without additional maintenance against dynamic data, responsive design variations, and cosmetic changes; and there's no dependency on detailed definitions by the developer. Therefore test cases are less likely to break during test cycles and between app releases. A test case can still be affected by the opening of a new window or a dialog box that appears in response to new input data, but the overall percentage of broken test cases is far lower than that of traditional automation techniques – generally in the vicinity of 5% to 10%. For regression testing, only test cases affected by new functionality need to be adapted or rewritten, typically around 20%-40% of all test cases. The sample cost of test maintenance and reuse with Jamo Automator is outlined below:

	Best-case scenario	**Worst-case scenario**
Number of test cases	60	60
Total days needed for automation	6	16
Percent of rework between test cycles	5%	10%
Number of days required for rework and maintenance between test cycles	0	2
Percent of rework between app releases	20%	40%
Number of days required for rework and maintenance between releases	1	6

Conclusion

Using a next-generation tool like Jamo Automator helps significantly reduce the time and effort required for creating and maintaining of automated test cases, compared to traditional automation solutions. The simplicity of script creation, combined with easy test script maintenance, resilience, and versatility makes Jamo Automator an ideal fit for DevOps, agile, and continuous delivery methods. You no longer have to wait 2-4 years to realize the ROI on your tools – returns can be achieved within a single release cycle of your app.

Jamo Automator's ease of use and visual interface allow for non-technical members of the QA team to take an active role in the test automation process. The automation effort can be easily scaled to keep up with release cycles and adapt to any organization's needs.

Jamo Automator is cloud-based, making deployment as easy as connecting to a URL through a browser. All your test data is securely stored on the cloud, accessible anytime, from any location.
To learn more, visit **Jamo Solutions.**

Sources:

a. Qingzhou Luo, Farah Hariri, Lamyaa Eloussi, and Darko Marinov. 2014. An empirical analysis of flaky tests. In Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering (FSE 2014). ACM, New York, NY, USA, 643-653.
b. Adriaan Labuschagne, Laura Inozemtseva, and Reid Holmes. 2017. Measuring the cost of regression testing in practice: a study of Java projects using continuous integration. In *Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering* (ESEC/FSE 2017). ACM, New York, NY, USA, 821-830.
c. Atif M. Memon and Myra B. Cohen. 2013. Automated testing of GUI applications: models, tools, and controlling flakiness. In *Proceedings of the 2013 International Conference on Software Engineering* (ICSE '13).