4th International Conference and Workshops on Recent Advances and Innovations in Engineering, 27-29 November 2019
(IEEE Conference Record # 47735)

Multiplatform Automated Software Testing: Personal Experience of a Maintainer

Maxim Mozgovoy
Active Knowledge Engineering Lab
The University of Aizu
Aizuwakamatsu, Fukushima, Japan
mozgovoy@u-aizu.ac.jp

***Abstract*—Automated testing is an essential part of modern software development pipeline. The extent of functionality to be tested varies a lot from project to project, but at least some basic testing capabilities are built into many current development instruments, and automated testing practices are encouraged in most guidelines. The goal of this paper is to share some personal experience with automated smoke testing of a cross-platform game application. Unlike works dedicated to the general picture of testing tools and practices, this paper focuses on specific details and challenges associated with setting up and maintenance of day-to-day automated testing activities.**

Keywords—quality assurance, automated testing, smoke testing, agile development

I. INTRODUCTION

Testing is an integral part of software development pipeline. Various types strategies of testing have been discussed in literature at least from late 1960s [1]. However, the practice of continuous automated testing as a part of daily routine, and recognition of testing code as a part of project's codebase gained momentum much later, and is usually associated with the "rediscovery" of test-driven development by Kent Beck [2].

Daily testing, like other agile development practices, is much easier to adopt with the support of specialized tools. For example, one may argue that while it is not overly difficult to setup a simple automated build server, the emergence of out-of-the-box systems like Jenkins or TeamCity has greatly contributed to the popularity of continuous integration practices. Gradually, typical testing scenarios also received external support — initially with testing frameworks like xUnit, and later at the level of major development environments (such as Visual Studio and IntelliJ IDEA) and automated build systems.

However, most readily available functionality is designed to support mostly low-level *unit testing*, while setting up testing of larger software components (*integration testing*) still requires much effort from the developers. This is not surprising, since integration tests are more project-specific, and it is harder to provide a truly universal testing framework.

Still, the last two decades were marked with the appearance of tools aimed to assist integration testing. For example, the process of testing of a website functionality can be greatly simplified with the use of Selenium WebDriver [3]. Thus, the task of setting up automated integration tests is perhaps less daunting nowadays than ever.

The author of the present paper has first-hand experience of setting up and maintaining an integration testing scaffolding for a mobile game project. The general overview of the system we designed and certain specific challenges we had to overcome are discussed in our previous works [4–6]. Here I want to focus mostly on the issue of *hidden costs*, i.e., on technical issues we faced while setting up the testing framework, and on regular maintenance activities, necessary for its smooth operation.

Such topics are rarely discussed in literature. They are often considered as "technicalities" not related directly to central ideas of proper testing organization. Still, it is useful to know potential issues that might appear in a project similar to ours and be prepared to address them. Admittedly, discussion of purely technical issues is like shooting at moving targets: many of them quickly become irrelevant as technologies grow mature. However, we can observe recurrent patterns in these issues, indicating potentially problematic areas.

Automated testing framework has become an essential component of our software development pipeline, so for us the benefits associated with autotesting outweigh the cost of efforts necessary to fine tune the system and keep it running.

II. FROM UNIT TESTING TO SMOKE TESTING

Testing strategy is one of the major topics to be clarified during project preparation. Naturally, under ideal conditions one might implement a procedure encompassing a wide range of automated and manual tests, ensuring conformance of the system as a whole and its individual components to specified criteria. In practice, however, certain tradeoffs are inevitable.

Agile development practices often emphasize the role of unit tests, designed to check individual functions and classes. Unit tests are typically automated and integrated into a continuous delivery pipeline. However, opinions on unit testing are divided. Some experts like Robert Martin see test-driven development (based on unit testing) as a strong methodology for producing *"clean, flexible code that works"* [7]. Others, like James Coplien, who calls most unit testing *"waste"* [8], are more skeptical.

Without going into the arguments of both parties, it is easy to observe that different personal experiences of professional developers are often caused by different *nature* of code under testing. In particular, Coplien admits that unit testing can be a sound strategy for procedural rather than object-oriented programs. An interesting attempt to divide code according to its "unit-testability" was made by Sanderson [9]. He divided code into four groups (see Table 1) with different costs and benefits of unit testing.

Smoke tests are often named as the most important tests to write, especially in case of severe time and cost pressure [10–

978-1-7281-2610-4/19/$31.00 ©2019 IEEE

12]. While basic smoke tests merely run the application and check whether the main screen shows up as expected [12], they can (and probably should) evolve into much more complex combination of core system functionality checks.

TABLE I. "UNIT-TESTABILITY" OF CODE

Costs and benefits of unit testing		Costs	
		Low	**High**
Benefits	**Low**	Trivial code	Coordinators
	High	Algorithms	Overcomplicated code

In our case, investing efforts into smoke testing turned out to be a very effective strategy. Smooth operation of a multiplatform online mobile game requires coordination of many distinct subsystems, responsible for backend communication, physics, animation, user interface and so on, and even minor flaws in any of them cause severe malfunctions, easily observable in simple test scenarios.

III. BASIC SETUP OF THE SMOKE TESTING SCAFFOLDING

Our testing infrastructure is based on Appium test automation framework[1]. Appium uses Selenium WebDriver API to provide uniform application testing capabilities for a number of platforms, currently including Android, iOS, Windows and Mac OS. Basic Appium setup consists of three components:

- a target device actually running the software under testing;
- a test runner device executing user-supplied test scripts;
- a test server, running Appium software and serving as a bridge between the test runner and the target device.

These components do not necessarily have to be installed on separate devices. For example, in case of Windows desktop applications testing, a single Windows machine can serve as a host for all three components. However, in case of mobile app testing, at least one mobile and one desktop device are needed. Furthermore, iOS automation is only possible with a MacOS-running desktop device.

Appium can be set up for operation in a hub/grid mode, where a special load balancer component chooses automatically the target device for running the next test script. However, for the sake of simplicity and higher fault tolerance we decided to run independent instances of a test server and a test runner processes for each target device. Thus, in our case, each new version of our game produced by the automated build server is tested on the device *i* by an independent combination

$$\text{test-runner}_i \leftrightarrow \text{test-server}_i \leftrightarrow \text{target-device}_i$$

Software components are hosted on three physical devices:

- a Windows machine hosting all test runner processes;
- a Mac machine hosting test server instances corresponding to iOS devices;
- a Windows machine hosting test server instances corresponding to Android devices and simultaneously acting as a test server and a target device for the Windows version of our app.

It must be noted that Appium provides only the capability to execute a certain single test script on a certain connected target device. It is developer's responsibility to integrate this functionality into the production pipeline, which might require considerable work. In our case, the resulting smoke testing system is able, in particular, to:

- retrieve fresh builds from the automated build server;
- run test suites;
- repeat tests failed due to target platform failures;
- generate detailed test reports.

I believe that a large part of this functionality can be implemented within a project-independent framework. However, to the best of my knowledge, there is no such framework available yet. Probably, the closest relevant project is Smartphone Test Farm [2] that provides remote control functionality for Android devices.

IV. POWERING A DEVICE FARM

While it is possible to connect Appium to target devices via Wi-Fi, the only stable and officially supported method requires a physical cable connection. Thus, for connecting a potentially large number of mobile devices to a single machine a USB hub will be required.

Running tests inevitably drains mobile devices' batteries, and the power supply is available only via the USB hub used. Conventional USB hubs on the market can be classified into "passive" (or bus-powered) and "active" (self-powered).

Passive hubs usually have few (3-4) USB ports and no dedicated power adaptor. In this case, power is supplied by the host machine and essentially shared between the connected mobile devices. Since a data USB 3.0 port of a computer can supply at most 900mA power current [13] (sec. 9.2.5.1), mobile devices will eventually discharge under heavy use (typical chargers supplied with mobile devices provide 1.0-2.5A output current).

Active hubs solve this problem by relying on external power supply. Unfortunately, most externally-powered hubs are not designed to provide simultaneous fast charging and data transfer capabilities. From our experience, it is very likely that a randomly chosen externally-powered USB hub will not be suitable for a mobile device farm. After a number of trials, we opted for *Plugable* 7-port charging hubs. At least for the mobile devices we use, they provide enough charging power to keep the batteries full during tests.

Even if hubs provide sufficient power, certain mobile devices might not be able to use it. This issue is not very common, especially for recently released phones and tablets, but we had experienced it twice: iPad 3 was unable to use full hub charging capability (so long-running tests often ended with a device shutdown), and Samsung Galaxy Tab E refused to charge from any data transfer-enabled USB port.

V. OTHER HARDWARE ISSUES

Conventional mobile devices are not designed to stay powered and run tests constantly. Thus, we expected to encounter certain hardware failures. Surprisingly, most devices so far exhibited almost no issues related to their hardware. We had to remove only one device, Nexus 7, due to a hardware failure (exhibited as sporadic shutdowns). However, being constantly connected to a power source is an

[1] https://appium.io

[2] https://openstf.io

issue for some devices. We experience regular battery swelling on phones of Xiaomi and Doogee brands, and have to replace the batteries every 7-9 months. Minor battery swelling is also visible on iPad Mini 2. Most probably, swelling in our case should be attributed to overcharging and lack of cooling [14], but the differences between devices working under the same conditions is remarkable.

VI. Choosing Devices

It is hardly reasonable for a small company to setup a device farm with a large number of devices. Additional devices increase overall costs without providing clear extra benefits. Thus, we decided keep the total number of devices in the farm under ten, and focus on diversity of their specs.

Our rationale for installing a particular selection of devices was based on several considerations:

- There is no need to install devices already in possession of our developers or testers.
- We should include devices with the lowest specs satisfying the minimal hardware requirements for the product. It would give us a chance to identify inadequate performance and out-of-memory errors early.
- Different versions of mobile OSes should be present.
- A variety of screen resolutions, CPUs and GPUs should be tested.

It is quite difficult to satisfy all these requirements given the abundance of different devices, especially in case of Android platform. However, there are much fewer mobile *chipsets* (CPU/GPU combinations) available on the market, as many different devices are built on top of the same chipset. Furthermore, it seems that the fragmentation is decreasing over time: for example, all major vendors are now retiring 32-bit architectures and non-ARM processors.

Thus, our strategy was to shortlist the most popular devices on the market and to make sure the game runs smoothly on them. We also identified the lowest-performing chipsets (according to publicly available benchmarks[3]), used in certain specific devices we wanted to support, and obtained them for the farm.

VII. Mobile OS-specific Issues

Many device failures, leading to false negative test reports, are caused by specific quirks of particular operating systems. We had no OS-related issues with the Windows desktop target platform, but mobile OSes required additional attention.

A. Notes on iOS Testing

Apple-produced iOS-based devices aim to provide uniform user experience, so we encountered no device-specific problems so far. The only exception is related to a certain issue in 32-bit iOS versions that caused failures in long-running tests until a device is restarted. However, since all 32-bit devices are now considered obsolete by Apple, this problem is not relevant anymore.

The most persistent recurring issue is caused by Apple's insistence on regular updates. Each time the next OS update is available, an iOS device shows the update confirmation dialog, blocking all incoming connections. Thus, a device with this dialog active will not respond to Appium server's requests until the user confirms the update or chooses "Later" option, causing the dialog to reappear later.

Turning on silent automatic updates for iOS might not be the best strategy, as new iOS versions sometimes break compatibility with Appium, and it often takes some time for the Appium team to catch up with these changes[4]. Thus, currently we opt for manual iOS updates, and install them only after Appium compatibility is confirmed. Another strategy would be to block automated updates completely. There is no official way to do it, but certain methods still exist. Perhaps, the easiest and most reliable one is to block all incoming traffic from Apple servers on the wireless router providing internet connectivity to iOS devices in the farm.

Comparing to Android platform, Appium/iOS interface is considerably harder to configure. In case of Android, Appium relies on Google-provided tool `adb` for automation. For iOS, a large collection of independent tools is used, and their configuration issues might be hard to resolve. In addition, software certificates, required to run apps on iOS platform, must be set up properly and updated when necessary.

It is also important to note that on iOS platform the testing framework is allowed to control only the application under testing, rather than access the user interface of the target device. It means that on iOS the tests cannot interact with system dialog boxes and change device settings (for instance, it is not possible to turn internet connection on and off to check how the application reacts to the loss of signal).

B. Notes on Android Testing

Android OS versions come in a large variety, but automation interface is very stable. The default Appium Android automation backend supports any mobile OS compatible with Android 4.4 or later. Android setup is relatively straightforward, and most issues we faced are device- rather than OS-specific.

One problem experienced on a large variety of devices is caused by unlimited growth of user data associated with Appium cache and/or `com.android.shell` system app. From the user's perspective it is manifested as steady consumption of available storage space until no new apps can be installed. We solve this problem by running the following cleaning script once a week:

```
adb -s <device-id>
    shell "pm clear com.android.shell"

adb -s <device-id> shell
    "rm -rf /data/local/tmp/appium_cache"
```

Several of our devices sometimes refuse to react to the "unlock screen" command sent by Appium. This can be solved by keeping the device's screen always on (achieved with the corresponding option in Android settings).

Some issues can be more subtle. For example, some of our tests type a certain string into an application's input box. During this process, all mobile devices show a pop-up keyboard. Some vendor-provided keyboards on Android devices do not react properly to Appium commands, and this

[3] Such as https://benchmarks.ul.com/compare/best-smartphones

[4] A notable delay was caused by a substitution of Apple's *UI Automation* framework with *XCTest* in iOS 10.

test ends with a failure. On such devices we had to install a third-party onscreen keyboard, compatible with our tests.

VIII. Appium Maintenance and Test Scripts

Appium consists of a number of loosely connected and independently developed components. From a QA engineer's perspective, the most important parts are *client libraries*, allowing to write tests in different programming languages, *backends*, responsible for communicating with specific platforms, and a Node.js-backed *Appium server*, serving as a bridge between test scripts and target devices. In addition, Appium depends on a variety of 3rd-party tools, assisting target platform automation.

While the project in general has reached a relatively stable state, individual components might cause issues that are sometimes difficult to investigate and overcome. For example, for months we had experienced sporadic Appium server crashes, presumably due to memory leaks. The problem was eventually resolved with an update to a newer Node.js version.

At least twice we were forced to update Appium for the sake of compatibility with a newly released iOS version. However, it turned out that the updated Appium server was not compatible with the newest Windows driver. Such incompatibilities between the newest stable and especially beta versions of Appium components, unfortunately, occur. Several times we had to use trial-and-error to find the right combination of versions of Appium components to achieve desired functionality.

One should also expect that certain testing capabilities mentioned in the Appium documentation might not be available due to limitations of the target platform, lack of support in a particular client library or a particular backend version. For example, Appium provides a way to simulate a tap/click in the specified point (x, y) of the specified GUI element. However, Windows driver did not support this capability till Sept 2018.

From the perspective of test script development, it is good to remember that mobile devices are not designed for stable test automation. Quite often a failed test initialization step (which involves waking up the target device, installing and running the app) will finish successfully during the subsequent attempt. Thus, test scripts should implement basic fault-tolerance measures, especially during these initial actions.

IX. Conclusion

Automated testing became an integral part of a modern software development pipeline. One of the most cost-effective types of automated testing is smoke testing, aimed to confirm the basic functionality of the complete system with a series of simple scenarios. Smoke testing is especially useful in cross-platform development, where manual testing imposes a considerable additional burden on team members.

Automated smoke testing can be implemented with the help of available third-party frameworks. The choice is more limited for mobile devices, and today Appium is probably the only real option for writing universal tests, executable on Android, iOS, Mac, and Windows.

In contrast to automated unit tests, smoke tests are harder to setup. With the present tools, it's still the developer's responsibility to organize the process of automated testing, including triggering tests for freshly available builds, load balancing, and reporting of test results. In addition, setting up a mobile device test farm can be a tricky process, fraught with numerous unobvious pitfalls. Certain investment of efforts into regular maintenance is also inevitable.

However, in spite of these challenges, we treat our personal experience as largely positive. Automated smoke tests facilitated early error detection and enabled our testers to concentrate on serious rather than trivial issues in the game. The landscape of mobile platforms became less fragmented in the recent years, while frameworks like Appium reached maturity, and are sufficiently stable now for daily use.

Personally, I hope that scaffolding for automated smoke testing will be eventually as accessible as software for file hosting or automated building. It will be greatly beneficial for the whole community of professional software developers.

References

[1] A. I. Llewelyn and R. F. Wickens, "The testing of computer software," in *NATO Software Engineering Conference, Garmisch, Germany*, 1968, pp. 7–11.

[2] K. Beck, *Test Driven Development*. New Jersey: Pearson Education (US), 2002.

[3] P. Ramya, V. Sindhura, and P. V. Sagar, "Testing Using Selenium Web Driver," in *2nd International Conference on Electrical, Computer and Communication Technologies*, Coimbatore, 2017.

[4] E. Pyshkin and M. Mozgovoy, "So You Want to Build a Farm: An Approach to Resource and Time-Consuming Testing of Mobile Applications," *ICSEA 2018*, 2018.

[5] M. Mozgovoy and E. Pyshkin, "Mobile farm for software testing," in *Proceedings of the 20th International Conference on Human-Computer Interaction with Mobile Devices and Services Adjunct*, 2018, pp. 31–38.

[6] M. Mozgovoy, "Quality Assurance in a Mobile Game Project: a Case Study," in *Proceedings of the 14th Central and Eastern European Software Engineering Conference (SECR'2018)*, 2018.

[7] R. C. Martin, "Professionalism and test-driven development," *Ieee Software*, vol. 24, no. 3, pp. 32–36, 2007.

[8] J. Coplien, *Why Most Unit Testing is Waste.* [Online] Available: https://rbcs-us.com/documents/Why-Most-Unit-Testing-is-Waste.pdf.

[9] S. Sanderson, *Selective Unit Testing – Costs and Benefits.* [Online] Available: https://blog.stevensanderson.com/2009/11/04/selective-unit-testing-costs-and-benefits/.

[10] G. Mustafa, A. A. Shah, K. H. Asif, and A. Ali, "A strategy for testing of web based software," *Information Technology Journal*, vol. 6, no. 1, pp. 74–81, http://www.docsdrive.com/pdfs/ansinet/itj/2007/74-81.pdf, 2007.

[11] Microsoft Corp., *Guidelines for Smoke Testing: https://msdn.microsoft.com/en-us/library/ms182613(v=vs.90).aspx*.

[12] J. Humble and D. Farley, *Continuous Delivery: Reliable Software Releases through Build, Test, and Deployment Automation (Adobe Reader)*: Pearson Education, 2010.

[13] Hewlett-Packard Company, Intel Corporation, Microsoft Corporation, NEC Corporation, ST-NXP Wireless, Texas Instruments, *Universal Serial Bus 3.0 Specification*.

[14] V. Challa, *Why Do Lithium-Ion Batteries Swell?* [Online] Available: https://www.dfrsolutions.com/blog/why-do-lithium-ion-batteries-swell.