Key Concepts in Test Maintenance

Perfecto

Seek Perfection

Test maintenance is more than just cleanup work... it's what you need to do to make sure your development cycles are completed efficiently and that your app works brilliantly. With an effective test maintenance strategy, you can maximize the impact of feedback from testing on early development cycles and fight technical debt at the same time.

An Effective Test Maintenance Strategy

All good automated software pipelines include testing, but don't always include well–defined inception and retirement criteria for their automated tests. When a test is created, there should be clear criteria for why and when it is created. Likewise, a regular audit of test suites should include formal test retirement criteria; without retiring tests, test suites would grow out of control.

Test inception criteria should include a connection to the business case[1] for the new functionality or workflow that the test covers. Since this is tied closely to the details of the feature or story, a common pattern in app development teams is to include UI testing as part of their own definition–of–done. Ultimately, the goal is to make sure that where new work creates coverage gaps, a decision is made to either log this gap as technical debt or resolve it with new tests immediately.

Before creating automated tests, the stability of the business requirements and app itself should be considered. "If manual execution is impossible, for example with an API, then any demonstrated testing and positive results will serve as a trigger for automation development," says Robert Galen in *Picking the Right Automation Candidates*[2]. Development teams which require automated tests for every new feature should protract the schedule for creating these tests as part of non–feature work for a later sprint when the app is proven stable enough to test.

[1] https://blog.testlodge.com/writing-test-cases-from-user-stories-acceptance-criteria

[2] http://www.compaid.com/caiinternet/ezine/galen3.pdf

Auto–generated Tests and GUI Recorders

Tools that help speed development are always a good thing, right? What about when they increase technical debt as you use them? It's not always clear when you start to use a tool that the artifacts it produces cause as much trouble as they overcome, but how do you know this ahead of time?

Recording tools that translate interactions at a point in time to test code more often than not produce code that is incoherent and incomplete. Though sometimes useful for UI inspection or initial test creation, these tools can easily provide a false sense of confidence that acceptance criteria has been properly imbued into test code.

Examples of popular recording tools include Appium Inspector[3], Espresso Test Recorder, and Selenium IDE. These tools can help to bridge skills gaps and form the basis of automated tests, but are no substitute for critical human thinking. As often is the case, the code produced by these tools must be altered before seamless playback can occur anyway, an indicator that these tools have limited uses and should not be regarded as a complete solution to creating reliable UI tests.

My point is: trust your development instincts. Learn to walk before expecting to fly. Write some code, learn the basics of a language or framework before trusting tools that magically whip out code for you. If you don't understand the output of a tool, be wary of using it as part of your toolchain. True confidence comes with practice.

For a great introduction to Espresso Test Recorder, listen to Android Developers Backstage episode #57 where Stas Negara discusses the ins and outs of the tool in detail.

[3] https://github.com/appium/appium-dot-app

When do I Retire a Test?

A test verifies that certain expectations are met. When expectations change (e.g. a screen is deprecated or replaced), tests based on those expectations need to be reassessed as candidates for retirement. Any time a screen, a workflow, or a major component is modified, it can cause tests to break. Broken tests cost time and, unless they are still relevant, should be flagged for cleanup. The time to do this is on every commit/push of these kinds of changes.

You may also find an alternative way to cover conditions that are already covered in an existing test. This doesn't mean that the original tests are irrelevant, since they may test certain portions of the app in isolation or under specific conditions. If however the smaller tests are no longer relevant, you may consider archiving them and only running the new tests to save execution time, provided that you are not reducing critical areas of coverage by doing so.

Execution Speed Matters

As testing technologies have evolved to meet development challenges, a key problem for many developers is the sheer time it takes to execute UI tests. With a combination of optimized frameworks and lab configuration, the modern trend with app teams is to specialize testing to multiple target platforms.

Emulators and simulators provide a fast and portable way to run these tests, either locally or as part of headless operation. They provide a wide range of operating system images, but can also present only a narrow slice of the complex reality that real users experience on mobile devices and in interactive web browsers.

When scaled to a continuous integration and build process, emulators often introduce numerous reliability issues, even under strict configuration management policies, due to their memory and CPU requirements. "At LinkedIn, we've even seen cases where the lock screen on the emulator is randomly triggered, causing tests to fail. This inconsistent behavior was causing our developers to lose trust in our tests and question why we were even writing them in the first place," says Drew Hannay of LinkedIn[4]. Emulators may be okay for real–time developer prodding, but still fail to live up to the always–on demand imposed by continuous integration.

Choice of test framework also plays a key role in the execution speed of tests. An example of this is to compare the execution speed of tests written for Appium to that of Espresso over the same Android application. Despite emulator startup times, wire protocol and execution model differences cause WebDriver–based test frameworks to be far slower than instrumented testing.

Finally, a key difference between instrumented and WebDriver–based testing is how UI delay is addressed. In many cases, arbitrary calls to 'sleep' cause test scripts to execute slowly despite a snappy, responsive app. Replacing sleeps with a polling pattern can quickly reduce overall test suite execution speed, but teams should still consider switching to an instrumented test framework wherever possible.

If you do use WebDriver, here are a few suggestions to speed up your scripts:

1. Minimize usage of XPath in preference to searching for elements by Id (Android) or AccessibilityID (iOS)
2. Minimize the number of places where you look for an element to be NOT FOUND, which avoids scanning of the entire element stack to prove non–existence
3. For Android, test on a real device. In most cases, this is far faster than incurring the additional overhead of an emulator.
4. Consider reducing the volatility of your tests (i.e. mock/stub out interactions with file systems, databases, and other apps) so that you have very few places at which you need to reset the device
5. If a value is used in more than one place in your script and doesn't change due to interactions, store it in a script variable instead of asking WebDriver for it multiple times

[4] https://engineering.linkedin.com/blog/2016/08/introducing-and-open-sourcing-test-butler--reliable-android-test

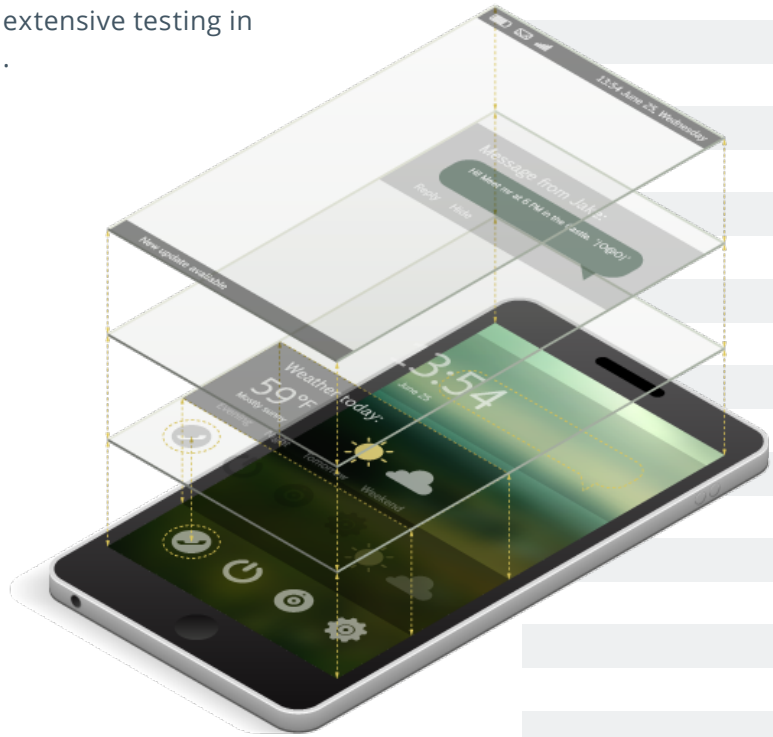
Hermetic Testing, Mocking, and Isolation

By definition, UI tests that include external dependencies will take longer than simple unit tests. So what can we do about it?

First, we can be as hermetic about the effects of our testing as possible. The Google Testing team says "A hermetic UI test typically runs as a test without network calls or external dependencies.[5]"

We can also "mock out" these external systems with components that provide pseudo–realistic results and responses that take far less time to return than actual app dependencies. Before jumping to full service virtualization, much can be done with additional frameworks such as Mockito and Dagger. For more information, see related articles[6] by Chiu–Ki Chan of Square Island.

In the end, you'll still need to make sure that the app is going to work correctly under real conditions. Isolation can be progressively relaxed in your tests based on the environment they are executed on. For instance, you can parameterize the API endpoints in your UI tests for each environment, pointing to local mocks for individual development activities, and pointing to a more realistic sandbox environment for more extensive testing in continuous integration.



5 https://testing.googleblog.com/2013/08/how-google-team-tests-mobile-apps.html

6 http://blog.sqisland.com/2015/04/dagger-2-espresso-2-mockito.html

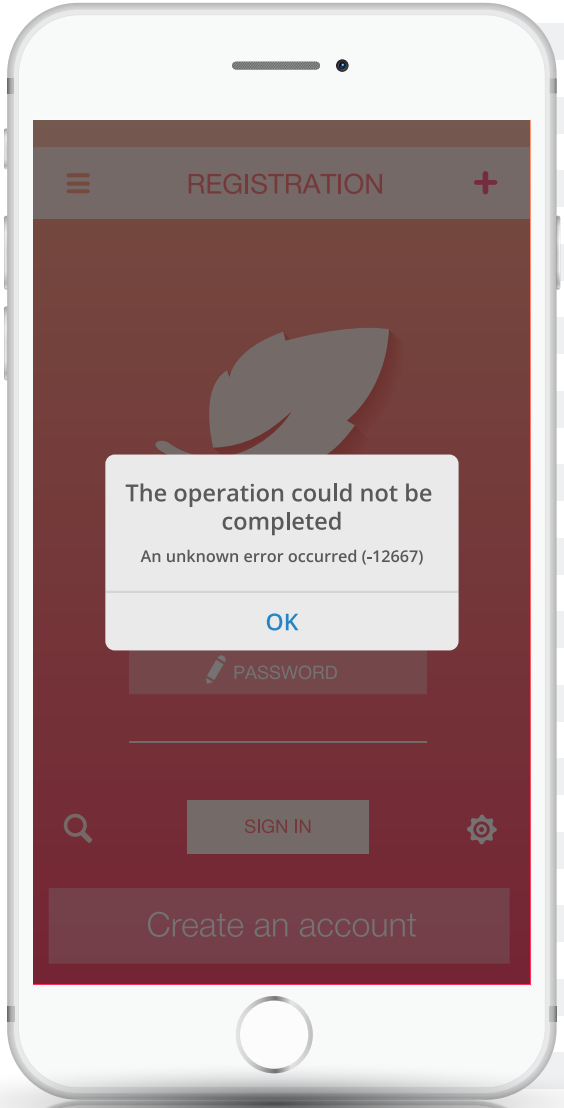
Improving Your Regression Testing Strategy

Without retiring tests, your regression suites would grow out of control. And even with the most neatly groomed test suites, full regression testing (especially with large data sets) may take longer than development cycles permit. The fact is that having an attitude that demands we "test all the things" doesn't scale and can undermine the effort of automated testing as a whole.

One approach to reducing time and risk in regression testing is to perform *less* testing *more often*... in other words, break your monolithic test suites apart into groups of tests that can be run continuously. While this doesn't remove the need to run full regression testing before releases, it minimizes the likelihood that defects will be caught late in release cycles, reducing last–minute thrash and freeing valuable time for app teams to work on things that improve the user experience.

A simple way to do this is to annotate test cases with metadata that helps to group them for execution on parallel schedules. Flagging a test as 'KeySmoke' vs. 'FullData' helps build and test managers run all key smoke tests on every build triggered by a new code commit, yet still schedule the full regression testing with data as part of a nightly schedule.

Other teams also use automated test impact analysis to identify which tests should be run based on their relationship to code when it is changed. Though not widely available for all platforms[7], various app teams often implement one form of this or another, such as using the results of unit test coverage to project impact on groups of UI tests.



[7] https://msdn.microsoft.com/en-us/library/ff576128(v=vs.100).aspx

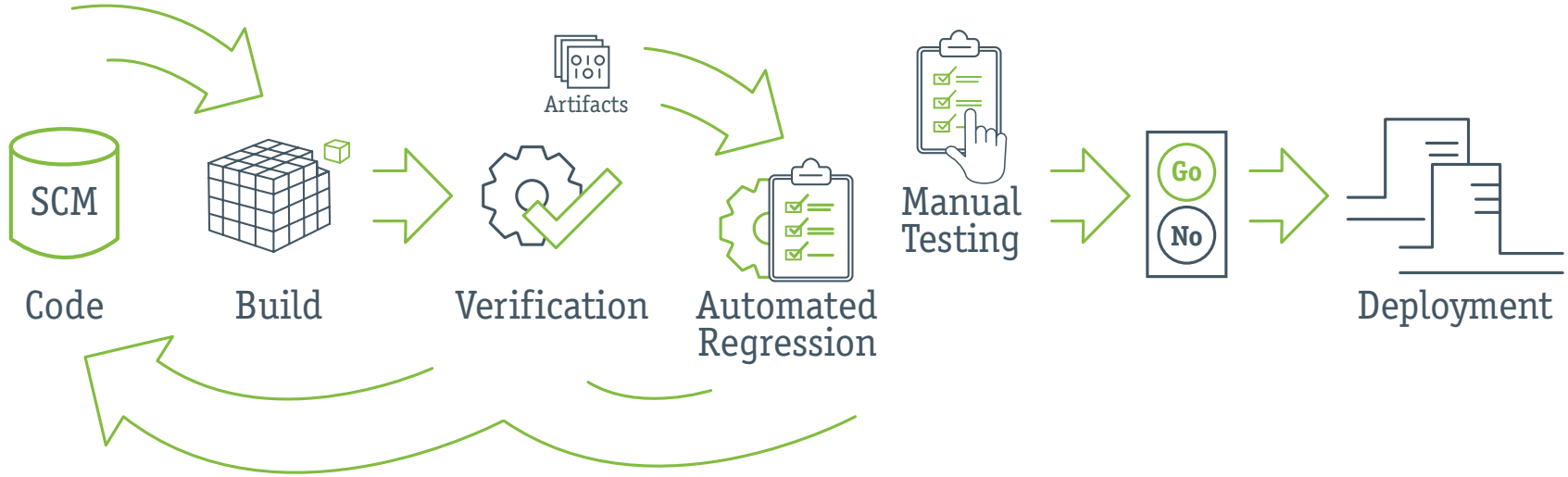
Parallel Testing to the Rescue

If you think your library of tests is big, consider how much larger the job is to run these tests on the wide variety of devices and platforms your app will exist on out in the real world! For each of these combinations, there are also numerous network and background usage conditions that make up the actual user experience.

The fact is that no amount of manual testing can cover the complexity we face today, but running our automated tests sequentially would take far too much time to meet our release schedules. This is where automated testing in parallel really comes to the rescue, shrinking our mountain of test executions to levels we can deal with in every build.

The simple math goes like this:

```
(total # tests) * (# unique platforms) * (critical user profiles) = total test
```



Obviously, due to the plethora of mobile devices in the market, both mobile and web app teams have a large number of unique platforms to test against, multiplying already large test suites by orders of magnitude beyond that which human or sequential testing can satisfy. Fortunately, programmable infrastructure such as cloud–based testing labs can help overcome this challenge, enabling us to achieve the necessary test coverage within our development cycles.

A great example of parallel testing with Appium is from the ThoughtWorks blog[8]. Taking it one step further, a better parallel test harness might include the following logical elements:

1. Initialize shared configuration (i.e. global variables) and credentials
2. Distribute the work to nodes
3. Track the status of work queue
4. Balance workload across resources (emulators, devices, browser sessions)
5. Consolidate results in real–time
6. Clean up all threads and resources used during the parallel test

While there are some rudimentary built–in mechanisms to parallelize UI testing under various frameworks (such as Espresso test sharding support in Spoon[9] or Perfecto's examples using Protractor[10]), specialization of this process per organization is often an inevitability and should be included in total project time and resource estimates.

8 https://www.thoughtworks.com/insights/blog/speeding-test-execution-appium

9 https://github.com/square/spoon#test-sharding

10 https://community.perfectomobile.com/posts/1129431-protractor-parallel-execution

Maintenance Takes Vigilance

I'm not going to lie; test maintenance isn't the sexiest or easiest job, but it is part of keeping technical debt down and keeping the delivery process up with the velocity of app teams. A few final takeaways to help include test maintenance into your app development team culture:

1. Every time you replace a feature or screen in your app, look for tests that fail afterwards and either fix them or flag them for 'technical debt' work in the next sprint.
2. Don't create automated tests for every interaction with every element in your app; instead, construct realistic usability cases that can form the basis of a small set of tests.
3. Chose the test framework that makes the most sense from both a development workflow and an execution management perspective; tests need to be easy to write and run.
4. If you write tests that change the state of the database, device configuration, or file system, it's your responsibility to leave these resources ready for the next person.
5. Don't write arbitrary code in your tests! Sleeps that have no connection to the actual state of the UI will cause more trouble in the long run than they solve in the short run.
6. Remember that emulators are easy for early feedback, but they have limited scope and your app deserves to be validated on real platforms before your user experiences it!
7. Cover the full list of target platforms and conditions that your app will experience in the real world; the more you know before release, the fewer fires you'll have to put out.

Paul Bruce

currently works with the Perfecto team as Developer Advocate, focusing on helping development groups improve their velocity and software quality throughout the delivery process. With more than 15 years of work as a full-stack developer over front-end apps and back-end solutions in non-profit, enterprise, startups, and small business IT, Paul now uses his experience to help other developers to save time, build cool and useful things, and encourages open source contribution wherever possible. Paul writes, speaks, listens, and teaches about software delivery practices in key industries worldwide and regularly publishes his research to blogs and engages online communities. You can read more on his blog at paulsbruce.io

About Perfecto

Perfecto enables exceptional digital experiences. We help you transform your business and strengthen every digital interaction with a quality–first approach to creating web and native apps, through a cloud–based test environment called the Continuous Quality Lab™. The CQ Lab is comprised of real devices and real end–user conditions, giving you the truest test environment available.

More than 1,500 customers, including 50% of the Fortune 500 across the banking, insurance, retail, telecommunications and media industries rely on Perfecto to deliver optimal mobile app functionality and end user experiences, ensuring their brand's reputation, establishing loyal customers, and continually attracting new users. For more information about Perfecto, visit www.perfectomobile.com, join our community follow us on Twitter at @PerfectoMobile.

Get content just like this delivered to your inbox!

Perfecto

Seek Perfection