Tri-Layer Testing Architecture

Péter Földházi Jr.
Quality Architect @ EPAM Systems
peter_foldhazi@epam.com

Abstract

Test automation engineers traditionally have designed their test automation frameworks with essentially two separate layers: Test Scripts and Test Libraries. This has been successful for some of them, but has shown to cause problems for many others.

The problem with this Bi-Layer architecture is that it does not allow the engineers to scale their solution to support multiple teams and especially multiple projects and organizations. As the application dependent libraries are intertwined, it becomes very difficult to reuse components for building new frameworks, to accelerate test automation in other teams and to scale test automation inside an organization.

That is how the idea of the Tri-Layer Testing Architecture was born: why not add a 3rd layer, for all the libraries that can be easily reused?

In this paper, I will talk about the main differences between the Bi-Layer and the Tri-Layer architectures. The three layers in the Tri-Layer Architecture are: Test Scripts (runnable test scripts), Business Logic (all the application specific libraries) and Core Libraries (independent, reusable libraries). I will provide examples from my experience at big financial technology, sportswear, online gaming and other companies, where I successfully implemented the Tri-Layer Testing Architecture and drastically lowered the test automation costs for these big companies.

Biography

I have been working at EPAM since 2012 and moved to the USA in 2019 where I have been working as a Test Automation Consultant and as a Quality Architect.

I have experience in mobile, web and desktop testing on all levels of test automation in the financial, gaming, fitness and other domains.

I am the first European and one of the first people in the world having successfully taken and passed the ISTQB Test Automation Engineer module exam. I am actively helping the ISTQB (International Software Testing Qualifications Board) through the Hungarian Testing Board by reviewing and authoring syllabi of foundation and advanced levels.

I am also an active speaker, having spoken at meetups and software engineering conferences such as STAREAST, HUSTEF, UCAAT etc. I used to be a guest lecturer at 3 Budapest based universities: Óbuda, Pázmány and the ELTE universities. Brewing beer and planting chilis are some of my hobbies.

1. Introduction

Test automation has become more and more prominent in recent years. While test coverage and reducing the number of issues leaking to production have been some of the key metrics for indicating the effectiveness of test automation; cost effectiveness, maintainability, scalability and modularity are also key aspects of a good solution.

That is why it is so important for test automation engineers to acquire architecture knowledge. Unfortunately, people tend to over complicate their diagrams, which will result in lengthy implementation time and higher maintenance cost. The concept of the Tri-Layer Testing Architecture helps address these complexity issues.

The three layers that I will discuss are:

- **Test scripts layer,** with the runnable tests
- **Business logic layer,** with the application dependent libraries
- **Core libraries layer,** with the application independent libraries.

2. Test Automation Architecture

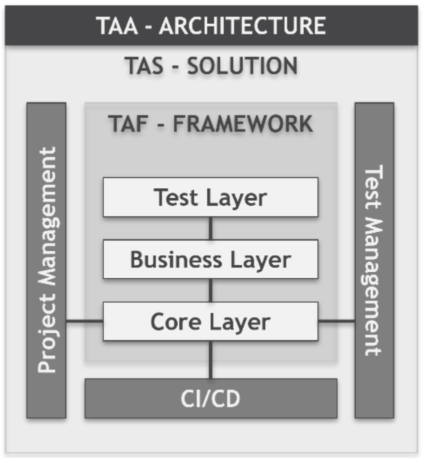
2.1 Framework expectations

Before going into details with the Tri-Layer concept, I want to make sure that readers are all on the same page when it comes to test automation architecture.

Let’s first think about what the typical expectations are from a test automation framework:

- Automated testing
- Detailed logging
- Generated reports
- CI/CD integration
- Easy to use
- Low maintenance costs
- etc.

2.2 Using the right terms in test automation design



Next, I want to talk about some of the terms that people sometimes confuse when talking about test automation design.

Test Automation Architecture (TAA) is the high-level architecture design of the solution we plan to build. It is basically a blueprint of that solution.

The Test Automation Solution (TAS) is the realization of the architecture. It includes the framework, the integration to CI/CD pipelines, the project management and test management tools as well.

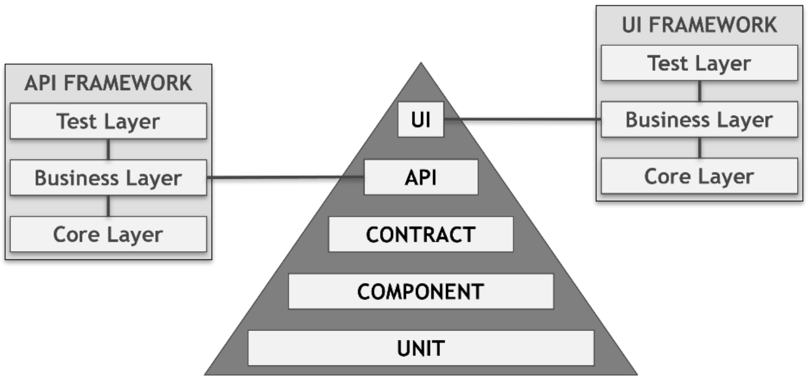
The Test Automation Framework (TAF) is the basis of the whole solution, without it, no tests would run. Therefore, it includes the test harness (typically JUnit, NUnit, XCTest etc.), test libraries, test scripts and test suites that are found inside the boundaries of the framework layers.

Framework layers define distinct borders of classes that have similar purposes. Your test scripts would fall under the very same layer. Page Models, Flow Models, an app dependent BasePage or BaseTest class file would make another layer that are application dependent.

2.3 Integration into software systems

Typically, simple frameworks have only one System Under Test (SUT). If you wish to support multiple applications or unrelated services with the same framework, the result is many times messy.

System level end-to-end testing involves functional validation of multiple applications, services, databases and their generated log files. The different levels should still be covered with different frameworks. Some of the basic libraries could and should be shared in the core libraries layer. If your only goal is end-to-end testing, then of course it is better to build just one all-purpose framework.



3. Bi-layer testing architecture

3.1 Linear scripting

One of the reasons why test automation projects fail is the lack of understanding of design concepts or the lack of any type of subject matter expertise in test automation. In such circumstances, companies typically come up with a simple solution that only includes test cases, where all custom test automation libraries are absent. Scaling and maintaining such solutions is quite challenging to say the least. Following this type of approach is not recommended in general as none of the elements of your test steps are reusable. Just think about a successful login flow. If you need to log in to an application in 50 different test cases, then you are essentially copy-pasting these steps into all the 50 tests. Whenever the login flow itself changes or the structure of the software under test, you will have to manually incorporate those changes in all the 50 test cases, instead of making that change in one place.

However, there are certain project setups or phases, when it is actually good to select linear scripting. Firstly, if you cannot afford bringing in test automation expertise, and only a couple of test cases need to be automated for a system that rarely changes, it is absolutely fine to create or generate your test cases without building an extensive framework.

Another case is when you wish to do a proof of concept before selecting the tool for building your test automation solution. You do not want to spend too much time on this activity, your goal is to compare a couple of tools and quickly make a decision, so that you can move on to building value for your team. Once the tool selection is done, you can start building a proper framework.

To give you an example from my professional life: a couple of years ago I was testing 2 iOS applications for one of my financial company clients. I wanted to see which test automation tool to select for building the test automation framework with. So I did a proof of concept (POC) with Apple’s XCTest UI library and with Appium. As I needed to by-pass the biometrics (TouchID), I selected Appium, as at that time only that tool was able to do the by-pass. For covering these POC activities, I didn’t need to build an advanced framework just yet. I wanted to first prove that test automation is possible, and then to prove which tool is better. In this case it was easy to do the selection, as only one tool remained by the end of the POC.

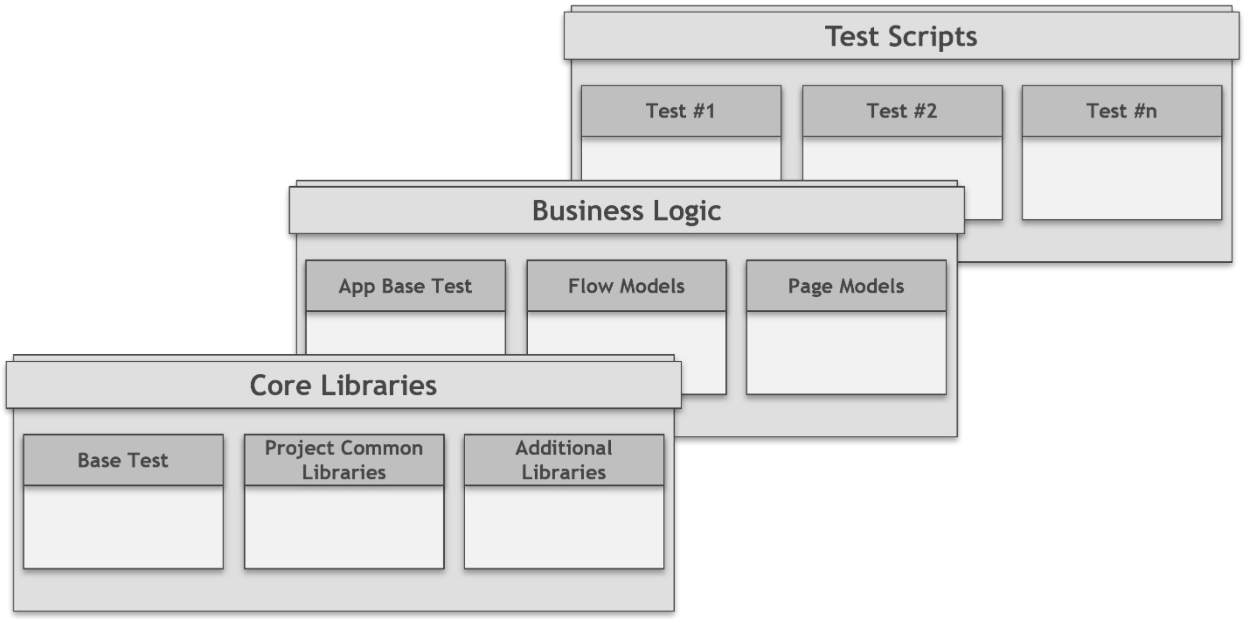
3.2 Structured scripting with two layers

Once we introduce libraries, it allows us to scale the solution and better maintain our test cases and data. In the linear scripting approach, we had only one layer: the test scripts. The bi-layer approach introduces the test libraries layer. This is where you find page models, flow models, utilities, step definitions, loggers, alerts etc.

This is already a decent architecture design to follow. It is maintainable and can scale much better than linear scripting does. But for bigger organizations, it is not scalable and there's circumstantial reusability holding back collaboration between teams and projects. By wiring in all the application and service dependent logic into your generic libraries (such as a Base Test, Utils or Alerts), you turn these libraries into app and service specific ones. Although it is necessary to have these configurations for your test solution to work, for others to reuse your libraries, they first have to decouple your product specific code snippets and only keep the generic ones. This may take days or even weeks to complete. And you also risk potentially giving away protected code or data that only your project is allowed to see.

In a DevOps and continuous testing world, this is a huge drawback. That is why I created the Tri-Layer Testing Architecture.

4. Tri-layer Testing Architecture



In 2013, I was working for one of the biggest sports clothing brands in the world. Initially, I was responsible for the test automation of one of their iOS applications. Eventually, I had to pick up the automation development for another related iOS app as well. That is when I came up with the concept of the Tri-layer Testing Architecture.

What I did was the following: identify the libraries that needed to be reused for building the framework of the second app. After the identification process, I did a pruning process in which I got rid of all the product dependent pieces of code and refactored them to make sure both frameworks will be able to leverage the same set of libraries. I started calling these common libraries as the core of the framework.

I moved the application dependent parts to new libraries that were inherited from the core libraries. This is what I started calling as middle layer at the time, but later I renamed it to business logic, as it made more sense.

Then I built a base framework for EPAM with only a core layer, which was then reused by multiple iOS projects' engineers.They were able to leverage this core layer to build their own framework on top of it. This allowed all our teams to kick-start their projects saving a couple of weeks of framework development work. The architecture design became easier as well, as instead of identifying 5, 6, 7 or even more layers (e.g. tests, feature files, steps, utilities, runner, reporter, logger etc.), my colleagues started identifying only 3 distinct layers, and that was enough on all of their projects. This again resulted in time saved. We were able to easily create maintainable automated test cases in the first couple of days of our very first sprint. We were able to mentor junior engineers to pick-up test automation knowledge quickly and also be productive during the first week.

Another example was when I designed an API test automation framework together with a test automation engineer colleague that he built for our financial client. Our goal was to implement a base framework that dozens of other teams could include as a dependency in their projects. Our business logic and test scripts layers served as the examples on how to build on top of the core layer, which was developed in a standalone repository. Once we demod the solution, all the teams that were interested in replacing their manual API testing efforts were able to include the base solution right away. This is a modular way of building your own framework. Think of it as playing with legos and you get to choose which building blocks you get to use. The same type and colour of legos exist, and everybody can reuse that exact same lego in their own lego building. Just like software libraries.

4.1 When to use and when to avoid using the Tri-Layer Testing Architecture?

Designing your architecture with this Tri-Layer approach is most beneficial when you plan to start your test automation project from scratch or when you plan to do a complete rehaul of your current solution.

If you are doing a re-architecture of an existing framework, then it takes some time even with a Tri-Layer design. Moreover, someone with architecture knowledge first has to do an assessment of the current solution and the system under test, to identify the scope of automation. Without a proper assessment, you will just create more mess, no matter how useful the architectural approach is that you are following.

In some projects, your team may be 100% certain that your test libraries won't be reused anywhere else and it's not that important to build a core layer right away. If that is the case, then it's totally fine to follow a Bi-Layer approach. However, the problem comes when managers do not want to give a couple more days for designing and building a core layer, and then it turns out that other teams' work could have been accelerated, had you created solution independent test libraries.

My suggestion for engineers is to still implement a core layer, as it does not take much more time to do so. In case in the future you do have to reuse your base libraries somewhere else, then you save some time for yourself or for a colleague. And if you can go open-source with your solution, then that's even better!

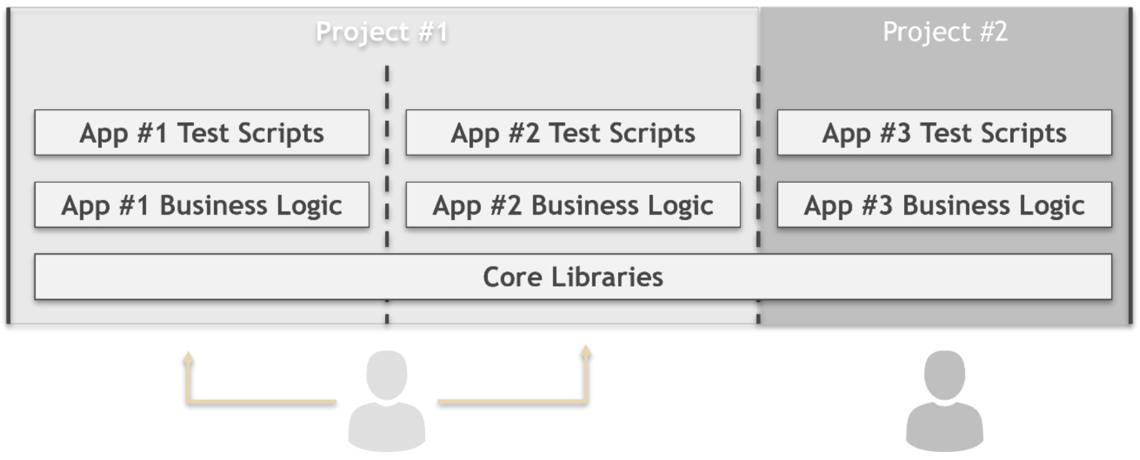
To sum it up, the test scripts layer stores all the runnable automated test cases along with the test suites. The business logic layer holds all the application dependent libraries, data and configuration. The core libraries layer consists of product independent, widely reusable components such as base test, base page, base validation, user actions and so on.

The core libraries may also include libraries that are specific only to certain projects, domains or technologies, such as a fake GPS location generator, gadget mocks, bank cards, cryptography, special app alerts etc.

5. Examples

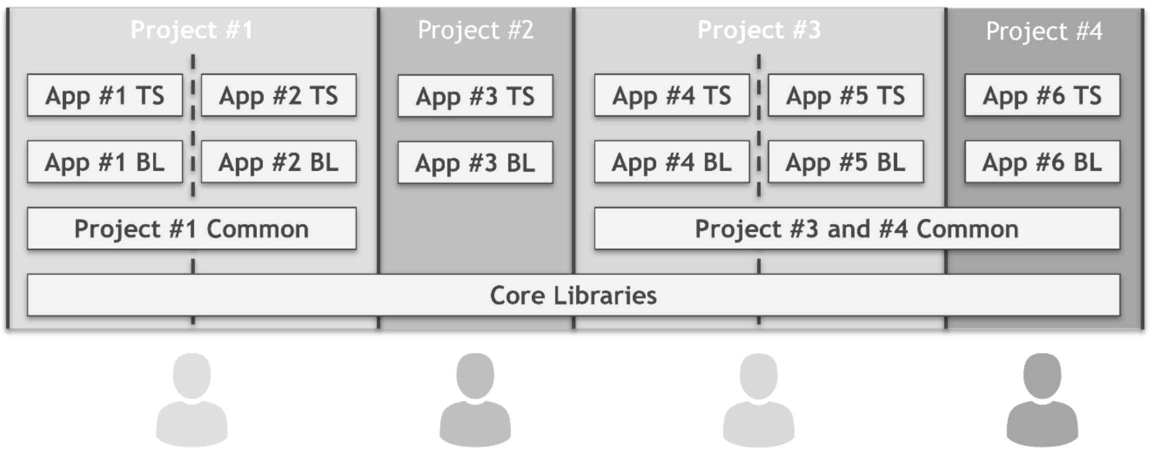
5.1 Example of reusability

This first example is identical to my first usage of the Tri-Layer Testing Architecture, when I built a solution for one iOS app, then I built another iOS test automation framework based on the core libraries. Later, other projects were able to reuse those libraries as well.



5.2 Project, domain and technology common libraries

In this next example, on top of the core libraries there are project-common libraries as well. Project #1 has its own libraries that are company specific (e.g. common way of handling users, or integration of hardware gadgets such as stride sensors). Project #3 and #4 share libraries that are domain specific: establishing connection to and processing data of public API’s such as locations on a map or the schedule of a city’s public transportation.

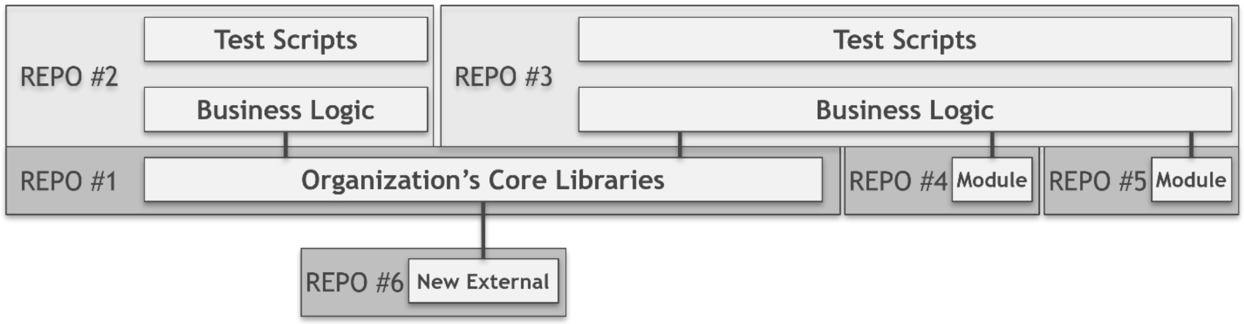


5.3 Repositories

The test scripts and business logic should be stored in the same repository. They could as well be in the same repository with the system under test.

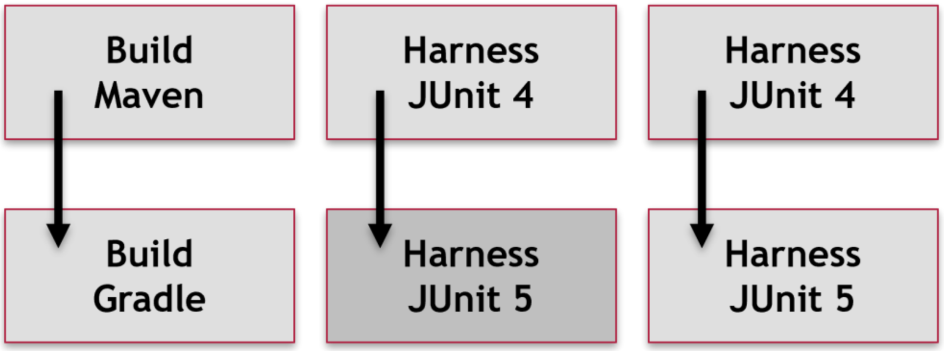
The core layer should be separate from the other two layers. Some of the modules that are considered as part of the core layer, can be managed in separate repositories. Typically, these are libraries that were developed by one team, then later picked up by other teams as well, but as not everyone want to sign-up to using these modules, they are not integral part of the organization's core libraries, but still part of the core layer for those teams that include them as dependencies.

New external libraries can be loosely coupled to the core. This allows easy replacement by pulling in an alternate, similar purpose library. External libraries are the ones that were not developed by the company's engineers. Typically, these are open-source solutions.



5.4 Modernization

As time goes by, some of the tools and frameworks will see major updates. In such cases, your organization could decide to adopt the newest version on organization level. If for any reason, that takes more time, then teams could individually decide to do the updates by overwriting the core functionality in the business logic layer. Reasons for not being able to merge such updates in a short time could be due to other teams not responding to change quickly. In such a case, updating it incrementally for each team makes sense. Eventually, all of the teams will be using the new version, and the dependency would be updated on core level instead of on business logic/team level.



5.5.1 Summary

The Tri-Layer approach provides a smooth design experience for those who wish to build test automation solutions that can be scaled easily, and which include libraries that are reusable for other teams as well. These advantages are present on project level, for the whole organization or even globally through open-source accelerators. This architectural design concept modularizes the framework into smaller libraries, which allows quickly building the framework itself and to easily update, replace or expand any of the incorporated libraries.

I hope that my readers will be able to improve their test automation after finishing this article.

References

Special thanks to:
Brian Crumrine, Gergely Ágnecz, Adam Auerbach, Philip Soffer, Andrew Pollner, Patrick Quilter, HTB (Hungarian Testing Board), EPAM North American QA Consultants

HUSTEF 2014 - Péter Földházi Jr. – Effective Mobile Automation:
https://www.youtube.com/watch?v=I2Y7VGI-iYE

ISTQB CTAL Test Automation Engineer 2016 syllabus:
https://www.istqb.org/downloads/send/48-advanced-level-test-automation-engineer-documents/201-advanced-test-automation-engineer-syllabus-ga-2016.html
https://www.istqb.org/certifications/test-automation-engineer

My personal website
https://www.peterfoldhazi.com