Structure, Design, Extensibility and User Experience - Foundations for an Automation Framework

Arvind Srinivasa Babu, Abhishek Sharma, Hemant Prasad
Arvind_Babu@McAfee.com, Abhishek2_Sharma@McAfee.com. Hemant_Prasad@McAfee.com

Abstract

Our team works on a native application that is deployed on tens of millions of machines worldwide. Frequent release commitments, new features and the agile process, stresses our need to increasingly speed up our validation process and our use of automation.

Over the course of the past decade, we developed couple of frameworks focused on features, services, or just simply automating our test suites. This worked well in the short-term but was unable to scale with our validation processes, which grew with every feature and release.

In this paper, we will highlight some of our pitfalls and how we overcame them by structuring our automation suite, segregating design, test suites, and considering the user experience in adding new test cases and extending the framework itself.

Biography

Arvind Srinivasa Babu is a Software Development Engineer and Software Security Architect at McAfee, with over 9 years of experience designing and implementing software. He leads the development of different features across server and non-server technologies while also building in security controls for the product. His areas of interest span network programming, user interface design and applications, cloud solutions, system programming, product security and mobile development.

Abhishek Sharma is a Senior Principal SDET at McAfee, Bangalore. He has over 9 years of experience in embedded as well as application software verification. Prior to McAfee, Abhishek led the Software Common Components verification team in Software Center of Excellence in Aerospace domain at Honeywell. Abhishek holds a Master in Computer System Engineering specialization in Intelligent Systems from Halmstad University.

Hemant Prasad is a Software Development Engineer at McAfee, Bangalore. He has 10 years of experience in analysis, design, and development of enterprise level products at McAfee. Hemant holds Master of Computer Applications from NIT, Karnataka, Surathkal. His area of interests is networking, system programming, and debugging and crash dump analysis.

1. Introduction

Automation[1] is the task of repeatedly performing a set of actions without any manual intervention. Automated testing is the principle in which the validation of software modules is performed automatically as new code changes are introduced in the software. As the software lifecycle keeps evolving, the critical ask for every software project is to deliver software faster and with highest quality. As per our observation over several years, automation has taken center stage in the domain of Software Quality as it drives faster delivery with highest quality confidence. While new greenfield projects have the advantage of developing automation as the new code is put in place, there are organizations that have had the struggle of moving legacy code from manual testing to testing with automation.

Our product is a native application that supports various platforms including Windows, Linux, MacOS, HP-UX, Solaris, AIX and all their flavors and versions with varying hardware architectures. Our software has been deployed over tens of millions of computer nodes with various network topologies and infrastructure. While it has been of paramount importance for our team to keep providing customers new features and product versions with the highest quality, we must also keep delivering updates with an ever increasing pace. Over the past decade we have had various challenges in automating the testing of our product and have employed various “Automation Frameworks” and have also developed our own.

In this paper, we will focus on our experiences in the past, what we have learned from it and what we have done to utilize our learnings. The paper is organized into various sections starting from where we were and leading up to where we are with our automation testing efforts. Section 2 will focus on our history, the challenges we faced in a non-automated world, how we incrementally set up smaller automation fronts, re-architected the product and along with it developed our initial home-grown automation framework. Section 3 will focus on how what constitutes an automation framework based on the learnings we obtained from our past experience. Section 4 will detail what we identified as foundations for any framework that needs to be developed for a product. Section 5 will discuss in detail on how to apply this Structure, Design, Extensibility and User-experience (SDEU) philosophy to a product.

2. The Challenges of Automating “LEGACY” code

Our product has been deployed worldwide in tens of millions of end nodes and it has been successfully adopted for more than 15 years. Our product also supports over 200 flavors of operating systems including Windows, Linux, MacOS, HP-UX, AIX, Solaris, etc. These metrics speak to the efficiency of our quality process, which has continuously provided a better customer experience. Our team has constantly changed in size in the past 15 years and averages between 10-15 engineers at any given time. This is the team that has designed, developed and validated the product across various process lifecycles, platforms and release timelines.

Before we embraced automation, all validations were done through manual testing. A small change in one of the core modules would easily render a manual validation effort of five to six months. We did have a few unit tests that barely provided us 3% functional coverage due to an architecture that needed to support multiple hardware platforms and operating systems.

Our initial attempt at automation was to increase our code coverage through unit testing for legacy code. While this certainly helped in increasing our functional code coverage by 10%, but it was still not enough. However, the core modules were actively tested through unit testing while functional verification tests and build verification tests were still being performed manually. This helped reduce a couple months’ efforts as core modules were not often worked upon and when we did work on them, the unit tests helped flag any anomalies with newly introduced code.

The above process worked well for a two to three year period, but a four month validation effort to verify a few lines of code changes was still affecting our delivery rate. We also started transitioning our lifecycle process from waterfall to a semi agile process since the validation duration was bottlenecked with manual testing. This was leading to story sizing issues as well as spikes in every sprint. We decided to tweak our process to fit our validation efforts. This was the time we decided to step in and build an automation framework, a home-grown project that could automate and validate test cases within the

python framework. This was keyword-driven automation suite to allow individuals to write new tests easily. It worked for a couple of years but as team members changed and new features came in, the framework couldn't scale up to meet the demand for new keywords and it eventually failed to enable team members to automate new features (Figure 1).

After being in the field for nearly a decade, we wanted sleek and manageable product code that could support various platforms and operating systems, so we decided to re-architect our product to position ourselves with embedded devices as well as the other platforms we already support. This also allowed us to move our architecture to service-oriented architecture and a new automation framework to specifically verify services. We mirrored our development of the new automation to our new service-oriented design of the product and added service verification tests to gain more coverage of our new services. While this new framework proved useful during the development of our product, as we approached our release milestones, we started to build security controls into the product. The automation framework failed miserably since it was not able to perform the validations it could during development and the entire effort done to build the service verification framework had to be dropped altogether.

The third iteration of the automation framework focused more on automation infrastructure resources and on enabling testing in a rigid environment where the automation code, test suites, test cases and infrastructure details resided as part of code. Although this rigid automation style restricts who can modify the automation framework, this mode of automation served its purpose for a few years where the validation efforts were reduced to a month with the automation focusing on positive test paths for feature validations that usually run to the better half of a day.

While our third iteration is no way perfect, it served us well for some time where team could avoid manual verification of basic functionalities and focus on verifying the new features being delivered. We identified the challenges and limitations in the automation framework over the period. The next sections will cover the identified challenges and how we overcame those while making our journey towards end objective of continuous integration and release cycle..

2.1. Challenges

We identified a lot of problems and challenges when attempting to automate our validation plans. A few of the challenges we faced are as follows:

2.1.1. Tools

There are a wide variety of tools to solve problems, which we face in validating products and solutions. The bigger challenge has been in deciding the best tools andfit it into our product's life-cycle process. As our product has been in the market for more than two decades, we realized that the most important factor is the amount of work required to adopt newer tools and frameworks.. This adds cost and complicates long term support of the product. Moreover, identifying commercial off-the-shelf(COTS[5]) softwares that continues to evolve and provide enhancement over the years, is a challenge.

2.1.2. Automation Development

As product grows, so does the automation suite/framework associated with it. Without proper maintenance or process it can quickly overgrow the product and become a complete mess. Building automation frameworks without a long-term vision for support and maintenance, the life of the automation framework is only as long as the developer who wrote the automation framework. It becomes non-reusable to more modern features and cannot support legacy features.

2.1.3. Design

A lack of automation design leads to quick fixes and short-term solutions but causes long term problems in maintaining the automation code. The automation suite/framework cannot scale or extend to support product engineering teams to validate future versions.

2.1.4. Code Structure

The most common approach we have observed is the quick fix approach to solve a problem or automate a component of the product. This leads to the framework code and the automation code for the business logic of the product component to be clubbed together, making the code non-reusable and rigid in its extensibility.

2.1.5. Software Builds

The idea of automation is to get results faster. A Continuous Integration (CI) process would result in giving faster results per check-in and can just run with unit tests, but the automation needs formally generated builds, which generally run into several hours depending on the language in which the product was implemented and size of the project. The delays in getting builds causes a lot of delay in finding out issues with newly checked in code, thereby causing overall delay in getting automation results.

2.1.6. Backwards compatibility

As product support runs long term, maintenance activities cause older versions to be supported with hotfixes or patches. The automation framework without versioning results in breaking older version validations due to addition of new code to support new/modified features. For example, a product is having a mainline development in version 4.0.0 and there are maintenance activities in its 3.0.0 and 2.0.0 versions. Automation codebase without versioning has been tuned to work with 4.0.0. If a hotfix or patch has to be delivered to 2.0.0 versions (as 2.0.1 or 2.0.x), he automation framework would miserably fail as the codebase contains automation code for 3.x and 4.x features.

3. What Constitutes an Automation Framework?

As we started to realize the challenges we faced in our past, we ultimately decided to ask the question “What constitutes an automation framework?” There is certainly a plethora of tools and each one of them solves a unique problem. But there is no tool that serves all purposes. We attempted several homegrown solutions to develop an all-in-one automation framework but as highlighted in the previous section, each attempt had brought about its own challenges. The other question that kept coming up was “For what purpose does the automation really bring value?” Should the automation framework focus only on faster delivery of validation results or should it also allow developers to be more vigilant about their code changes breaking something. We had unit tests running in our CI part of the pipelines, we also had a mix of tests automated for end-to-end workflows and the rest was covered as part of our manual validation efforts. So how do we keep things simple and automate our validations across white-box, grey-box and black-box testing as efficiently as possible?

We started to derive the foundations from what we have learned from our past encounters and we used that to build the foundations for the new framework. The automation framework will exist in any product that wants to automate their validation process. We define an Automation Framework as a codebase that controls, monitors one or more tools and independently facilitates execution and verification of test cases.

4. Foundations for an Automation Framework

We believe that any Automation Framework that is being developed should be founded on these four critical principles. An Automation Framework should encapsulate:

- Structure
- Design
- Extensibility
- User Experience

4.1. Design of an Automation Framework (D)

Any product being developed requires a simple but robust design to allow more features to be built into it. The same principle applies when designing an automation framework. It is essential for framework developers to model the framework in such a way that the product can exist for decades. Designing a framework to solve a short-term problem will give a life-expectancy for that short-term only. Most of our previous iterations in designing the framework aimed at solving short-term problems and that framework lasted only a few years.

A strong design for an automation framework should

- Be modularized.
- Be able to handle automation of standalone test cases as well as test scenarios requiring complex configurations and setups.
- Abstract the underlying tools employed to validate a test case.
- make it easier for test automation engineers to add new test cases.
- Allow rapid integration of new as well as old technologies or frameworks from other scripting languages.
- Be extensible to accommodate multiple component architectures.
- Be able to integrate with continuous integration tools such as Jenkins[4] or Teamcity[3].

A strong design is what allows developers of products to keep building new features easily into a product. The same principle is often overlooked when writing for automation. A robust design provides inputs when structuring your framework, as well as provides extensibility and simple user-experience to write product test case automation.

4.2. Structure of an Automation Framework (S)

When a product is developed for more than a decade it shouldn't be surprising when you end up with millions of lines of code. This would lead to your automation code as well to grow into a complex framework. The structure of your Automation Framework is defined as the boundary within which your automation code can be segregated. We believe this is one of the most important aspects to consider when writing the framework as well as when writing the test cases.

Here are some few pointers when defining the boundaries for your framework.

- Never combine product test cases with framework code.
- Apply versioning to your framework so that users know exactly which version of framework has what capabilities.
- Define packages (if applicable) to organize your framework code. Monolithic framework requires high maintenance.
- Maintain separate repository for your product test case automation code, if possible in the same branch as where the product development happens.
- Identify product features and their interdependencies and model your product test cases to mirror the dependency matrix of the features.

Maintaining a structure discipline in your framework and test code allows your framework to scale and be maintainable as your product's features grow.

4.3. Extensibility of an Automation Framework (E)

Any framework that is being developed should have provision to extend its capabilities at any given point of time. A framework that is designed to be rigid will ultimately solve the short-term problem but will fail on the long-term. A few items to consider for extending your framework:

- Allow addition of new features to framework.
- Allow addition of new product test cases.
- Allow complex reporting mechanism on success and failures.
- Allow addition of new programming languages.

- Allow addition of new tools or substitute old ones.
- Minimize or avoid rewrite of product test cases.

4.4. User Experience (U)

Each of the below user profiles will have different expectation when using the automation framework.

- Automation Framework Developer.
- Test Case Automation Developer.
- Product Developer.

4.4.1. Automation Framework Developer (AFD)

This user is primarily responsible to add or maintain the framework and provide integration and reporting mechanism for other types of users. A good experience for an AFD can be characterized by

- Ease of maintainability of framework codebase.
- Ease of addition or removal of feature support.
- Ease of addition or removal of external tools.
- Abstraction of product test cases from framework codebase.
- Versioned framework releases to track and maintain framework capabilities.

4.4.2. Test Case Automation Developer (TCAD)

The predominant user of an Automation Framework would be the product's test case automation developer. This can be any quality assurance engineer including the automation framework developer themselves. A good experience for a TCAD would be

- Ease of integrating test cases into automation framework.
- Platform agnostic way to acquire infrastructure resources to perform validation.
- Ability to write automation code that is coherent with the manual test case execution workflow.
- Framework that allows multiple technologies or architectures when automating different features.

4.4.3. Product Developer (PD)

The product developer is primarily the initiator of new product features and their inputs might be essential in adding new framework features to support automation of test cases. A good automation framework experience for a product developer would be

- Faster results of unit tests.
- Faster results of product feature validation.
- Faster results of end-to-end testing.

5. Applying SDEU philosophy to non-server technologies

While the Structure, Design, Extensibility and User-Experience (SDEU) philosophy can be applied to any product or automation, we will be focusing on applying this to automation of non-server technologies.

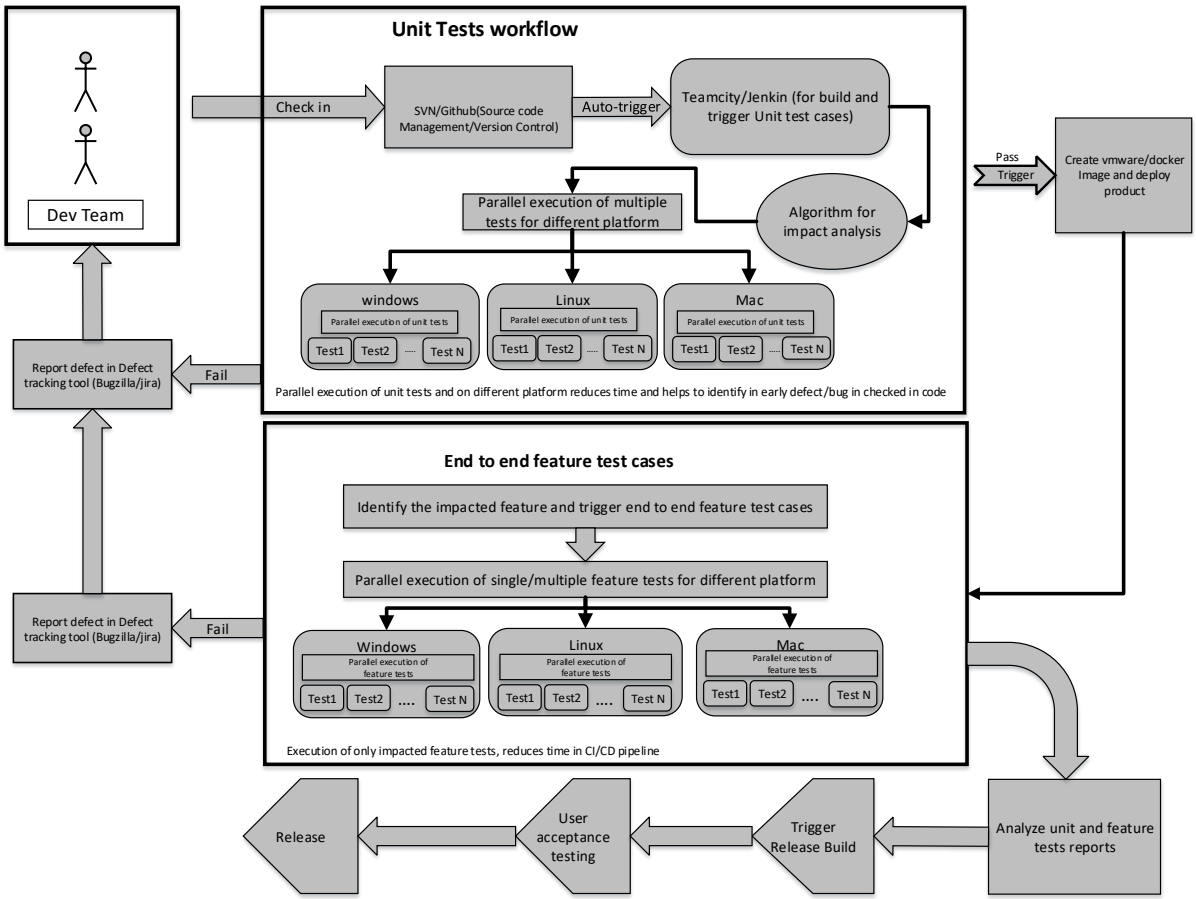

Figure 1- Overview of apply SDEU on an Automation Framework End-to-End.

5.1. Identify the structure of the framework

- Gather the infrastructure requirements to perform validation of your product.
- Design the framework for reusability. Make components for your framework that can be re-used everywhere. Group utilities and build your framework on top of that.
- Identify your Continuous Integration (CI) pipeline and how to fit your automation to make Continuous Delivery (CD) of your product.
- Define an integration mechanism to allow automation engineers to seamlessly automate their product.

5.2. Differentiate test cases from the framework itself

- Never combine the automation framework with product test cases. The framework needs to be agnostic of the product to maximize reusability.
- A product test case will entirely focus on the product, whereas the framework would abstract the complexity of management of underlying tools.
- If possible, maintain versioning of the framework to facilitate addition of features and updating of tools as and when necessary.

5.3. Optimizing build process

While the build time of a product is entirely dependent on the technology or language used to develop the product, it does take considerable time and delays to validate code changes leading to duplication of a CI process and a build process.

Generally, when a build is triggered, debug and release configurations of the product are built sequentially. While there are no relations between the two configurations and can in fact be built in

parallel, a small optimization is to modify the build script to build the configurations in parallel. (A MS-Build script can allow parallel jobs to build Visual Studio solutions or adding -j in a make file can produce a significant difference, depending on the size of your product and dependencies between its various components.)

5.4. Segregating white box, grey box and black box testable code base

Not every line of code written in the product needs to be validated through black box manual testing. Blocks of code that can run without any resource dependencies like server setups etc. can be unit tested.

- Identify areas of code like utilities or helper modules that can be unit tested independently of the product's core business logic.
- The business logic may contain some blocks of code that can be unit tested, but based on our experience a lot of code changes that happen in this layer of a product's architecture are better grey box tested Most of the automation code will be written for this layer.
- The end-to-end system verifications tests, or integration tests of our product from deployment to uninstallation falls under black box testing. This part of testing also needs to be supported from a framework perspective.

Having a clear demarcation between our product's utilities, platform support matrix and business logic helps build a reliable framework.

5.5. Keep Unit Tests simple

The purpose of unit test[2] is to test whether individual units of source code or modules with associated data or procedures are fit for use. The definition of "unit of source code" varies from individual to individual and on the implemented programming language. In C or C++ development of our product, we have had debates on defining units of code, for e.g. a non-static function in a .c or .cpp file or exposed functions through DLL. In Java, it is the public methods of a class.

Based on our experience, it's impossible to keep adding, updating and deleting unit tests for every piece of code written during development when features are added, modified or removed. To keep things simple, any non-business logic of your product needs to be unit tested and the tests need to be updated accordingly, based on the code change that goes in. Usually utilities, helpers, configuration of files and folders fall under the unit-testable category.

In C/C++, macros allow unit testers to make hidden static functions exposable based on some preprocessor definition at build time to expose the function to be unit tested. This allows unit tests to cover almost 100% of the code. Executing these unit tests falls under automation framework purview and the framework needs to expose methods to integrate new tests.

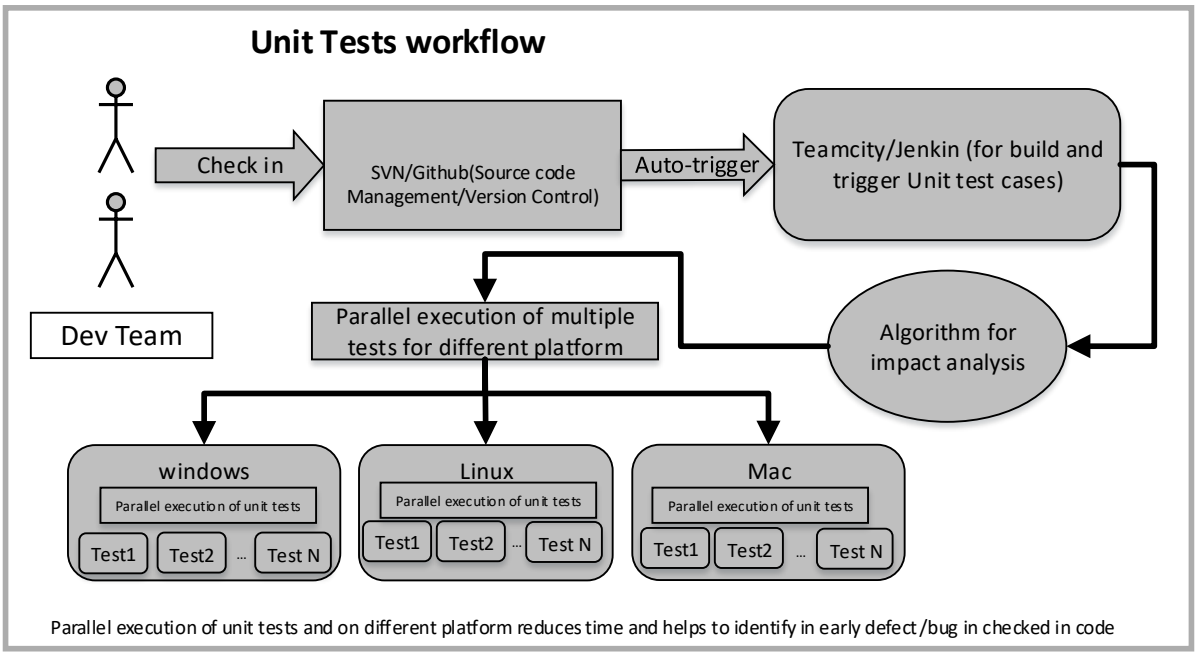

Figure 2 – Unit Testing in the Continuous Integration Pipeline

5.6. Identify tools for grey box and black box testing

A single tool or script written in one of the popular scripting languages like Perl or Python might never solve automation efforts for all features. The design of our automation framework should be accommodating in including different tools and provide a seamless switch between different tools and help in configuring them.

A product is made up of hundreds of features. Some of them may be independent and others interdependent on other features. As an automation framework developer, it is essential to identify the best tool to collectively test a particular feature or group of features and build the tool into the framework. This essentially granulizes the framework itself and helps to maintain regression tests for features.

For system level tests and end-to-end tests, infrastructure orchestration, tools to deploy the product and run integration tests if our product is part of a larger solution, should be taken into consideration when designing our framework code.

5.7. Defining pipelines for components and features of the product

A pipeline in an automation framework is something we define as the process of setting up all necessary infrastructure and resources to validate a sub-component or feature of the product. Splitting up the product into features and related group of features helps in maintaining a strong automation pipeline to validate code changes and perform regression testing when required. The granularity of the feature allows selective validation of just the feature(s) being impacted by code change.

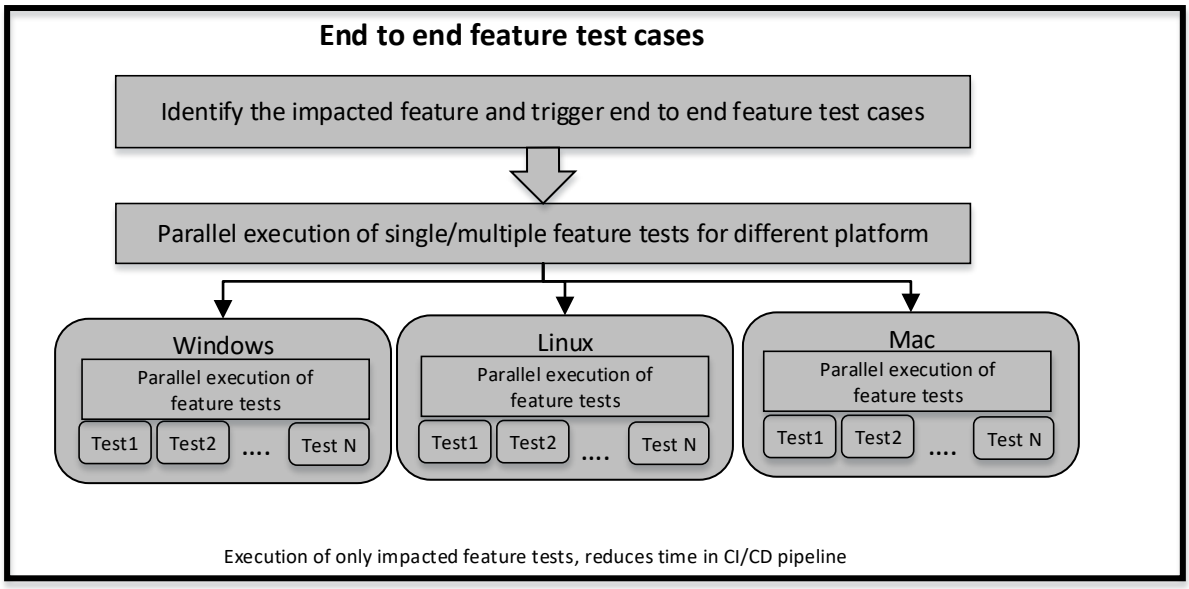

Figure 3 – Feature Verification Tests in a Continuous Delivery pipeline

5.8. Defining end-to-end automation pipeline

While a feature pipeline helps in validation of existing and new features in the long run, integration tests and system tests deal with a whole other infrastructure and resource complexity, which should be run independently and at regular intervals (maybe per day once or twice a week depending on the build time of the product). The inputs for building an end-to-end automation framework would rely on parameters and configuration information collected and tests modeled from at least one round of manual testing.

5.9. Framework implementation to connect the dots

With all the different components of the framework in place and the tools necessary to run our automation suites and tests, the framework implementation should focus on two main areas.

5.9.1. Abstracting the underlying tools

For a TCAD engineer who engineers the test cases into the framework, the seamless experience would be where the engineer need not worry about what tools to use and what not to use. The framework's integration kit would hide away the complex details and that is one of the focus areas when implementing the framework. For e.g., to execute unit tests, the framework would hide away the CI server like Teamcity[3] or Jenkins[4]. The TCAD engineer or Product Developer simply integrates the unit tests to run with the Framework's integration kit and the framework would take care of executing it in a Teamcity CI agent or Jenkins agent.

5.9.2. Make extending the test code developer friendly

For a TCAD engineer, writing framework code as part of product test cases should be avoided and the roles and responsibility of a TCAD engineer should be solely to automate the test cases for the business logic of the product. Any and all code that is required to ensure the test case can be easily automatable is brought under the framework implementation and the functionality required is exposed via the integration kit under a suitable package or component. This allows both the Automation Framework Engineer as well as a TCAD engineer to keep a productive pace in developing more test cases and build on top of the framework.

5.10. Automate away!

With a clear segregation of different areas of the products validation areas, separate repositories for framework and product automation code, pipelines being set up for regression of features, system and integration tests gives a holistic perspective to automation engineers keeping a maintainable source code that is easy to scale and extend.

6. Conclusion

We briefly went over the challenges faced when developing an automation framework from scratch and our inferences from the challenges. We formulated a philosophy that can be built into any product line or technology based on the structure of an automation framework, its design, and focus on it extensibility, all culminating in a seamless user experience to automate test cases. After applying this to our product, we have had significant improvement both from a unit testing perspective and automation codebase. We were able to improve our unit test coverage to 27% conditional focusing on the areas demarcated as utilities and helpers. We were able to set up pipelines to validate our install and uninstallation steps and other core business logic of our application and run system level automation validations. While there is still a lot of work ahead of us in converting our manual regression tests to automation code, we have made it simpler for anyone in the team to pick up and automate the product's test cases.

References

1. Test Automation : https://en.wikipedia.org/wiki/Test_automation
2. Unit Testing: https://en.wikipedia.org/wiki/Unit_testing
3. Teamcity: https://www.jetbrains.com/teamcity/
4. Jenkins: https://jenkins.io
5. COTS: https://en.wikipedia.org/wiki/Commercial_off-the-shelf