SMARTBEAR

TestComplete | White Paper

7 Steps for Building a Successful UI Automated Testing Framework

What are Automated Testing Frameworks?

Testing frameworks are a set of guidelines, or rules, used to create and design test cases. They provide a standardized test language and reporting structure for applications under test and can reduce test maintenance costs. Frameworks will link tests back to other parts of the software development lifecycle (SDLC), such as requirements and defects, enabling teams to find and fix bugs quicker.

Not all testing frameworks however, are automation frameworks – something vital to keep in mind when developing the latter. A testing framework is comprised of all of the

tools and practices intended for application development, from requirements outlined for your application, testing activities including both manual and automated tests, to environments to run your tests on, and so much more.

Automated testing frameworks focus specifically on optimizing automated processes. They facilitate faster cycles through test reusability and through speeding up test creation and maintenance by separating test data from logic. There are many types of automated testing frameworks, so it's crucial you adopt one that is right for you. Utilizing one that is well-structured however, can increase your team's efficiency by improving test accuracy, maximizing test coverage, and lowering costs and maintenance – ultimately giving you a higher return on investment (ROI) for your efforts.

Follow the next seven steps we lay out in this white paper to build yourself a robust automated testing framework and set you and your team up for long-term success.

Building a UI Automation Framework

1. Structure, Organize, & Set Up Source Control

Start by setting up and organizing a folder structure for your test assets. You'll want to keep different assets separate from each other, such as tests, name mapping criteria, and scripts, and create the files you know you'll need within each one. For example, within your 'Scripts' folder, you'll want to create files for each type of script – event scripts, actions, utilities, and verifications. Also make sure to create a file for your data.

Structuring your assets in this manner will allow for them to be quickly referenced by members of your team and will ensure your tests are stable when updates are made. When structuring your assets in this fashion, you can revisit the project at any time without the pain of needing to sort through a lot of information. This will also templatize your test folders, enabling you to clone them across a project.

When walking through these first steps, ensure you're using a source control management system (SCM) like Git or Mercurial to store your work. In the event a mistake is made, you won't want to lose that work or time spent. Tools like the will allow you to backtrack if needed.

2. Familiarize Yourself with the Application

Step two is to start familiarizing yourself with the application – beyond what the requirements can tell you. Reading documents that outline what the app should and shouldn't do will only get you so far. In this step, you want to get your hands dirty.

Conduct exploratory testing to give yourself an idea of how the internal workflows of the system are set up. This exercise will ensure you know how the application works. Once you've done this, you'll need to create a system for, or update, how you're finding your UI objects. Depending on the tool you're using, this could mean you need to create basic name mapping properties, or write scripts for the actions needed to identify objects.

The key to this step is to record your actions. Take notes on what your requirements will be and how your automation assets will test them.

3. Determine Your Testing Environments & Gather Data

Next, gather the data you plan to use for the tests and set up your environments. It's vital to your success that you set up configurations that can be run in more than one environment. It's time to accept that event handlers are your new best friend.

An event handler is a function comprised of code, that acts as a listener, waiting for an event to happen to trigger a script, or a series of scripts. Imagine your average banking application. After a certain number of minutes of idle time, a notification will pop up asking if you need more time. If you don't respond, it will automatically log you out for security reasons. Your event handler here is the function that triggers the scripts for the idle notification and the logging out process. Regardless of which environment you're on, maybe Windows 10 or Windows 7, you'll want this workflow to happen, and your end user would expect it too.

Event handlers will enable you to complete actions that respond to an event, without the need for a separate set of tests for each environment. They are the instructions for how a system should run, without the specifics, and will allow you to add sophistication to your tests without having to manually handle them. For example, you'd be

able to change any dynamic object identification properties, such as URLs or file names.

If you change the name of your application and need to point it to a different path, or update how it's installed, your framework will be able to address this with the help of event handlers. Tests are used to ensure pieces of the application behave as expected, and framework provides the tools do so.

Now onto your test data. Your framework should house the data separately from the tests. Use your repositories to store data and keep your properties and references generic – not test specific. This will allow your data objects to be shared between scripts and your data to be used across all of the utilities you have saving you time and effort in the future.

4. Set Up a Smoke Test Project

Before you create your utilities and verifications, it's crucial you get a smoke test project set up. They will become the most important set of tests you will use to verify your utilities.

Smoke tests, or build verification tests, validate that the most vital functions of an application work as intended and determine whether additional testing is needed. If a smoke test passes, it means the vital pieces of your application are working, so you can go ahead with more in-depth testing. If it fails, it means the basic functionality your application that needs to work is already broken. When this happens, you're better off asking for this to be fixed first. Further testing at this point would only be wasting precious time.

As your software matures, or expands capabilities-wise, your smoke test suite will also need to grow. It takes only one bug to cripple an application and diminish a company's reputation.

5. Create Utilities for On Screen Actions

After familiarizing yourself with the application, gathering data, and setting up your environments, you'll need to create shared utilities for common user interface (UI) actions such as menu navigation and text input fields. These are the basic building blocks of your tests, which you can then piece together to form the test logic. Depending on the tool you're using, this could be as simple as dragging and dropping pieces into a keyword test. This will allow your framework to drive your test flow and verifications so that minimal maintenance to individual tests is required.

A trick here is to use JavaScript classes within your framework, especially for navigation. This way, if you update how you're logging your actions, or your expected test results, you will only have to do this from one location – ultimately keeping your logging consistent.

This will enable team members who aren't automation engineers or developers to look at your test logs and understand what's going on. They will be able to determine if failures are an asset, problem, or an actual defect in the application, which is why it's important to abstract your framework data from the actual test data.

6. Build and Manage Verifications

In the next step, you'll want to set up your verifications - applying the same logic from how you structured your data, meaning they should be shareable. Let's say you're testing the functionality of your application and the requirements change on a text field. If you're verifying that the text field only accepts numeric characters and not text, your tests will need to be updated.

In instances such as this, you don't want to get stuck updating every single test, especially if you are still doing this manually. Ideally, you update the verify portion of your text field in one spot so that you could have 50 tests that test several different scenarios still work. Any UI verifications you build into your actions should be optional, so that in the event a field accepts an input properly and the test passes, there is no need to verify that action every single time.

Your verification data should also be shared. Different input utilities should be able to accept data objects created in previous steps so you can chain items together. This will also enable you to make updates in one area when needed, which can then be propagated across all the different areas of your framework – highlighting again, the importance of separating framework data from the actual test data.

7. Set Up Your Logging & Reporting Mechanism

The final piece of your UI automation framework is your logging and reporting mechanism. Throughout the entire build process, you should be recording and taking notes on all your exploratory actions, data preparation, and environment and verification building. Log messages before verifications, stating what it is you're verifying and what the expected result is. You'll want to make these messages human readable so that the non-technical users can look at your log and know exactly where and why a failure occurred. Errors should not be cryptic and no one should have to guess why they happened.

The goal of this step is to help you standardize your process – what your logging, when you're logging, and the errors.

Reporting should also be automated. Automating reporting will cut down on the time you spend pulling reports and enable you to focus on analyzing the data. If you want to export your test logs to share across your network or on a web server, with the right tool, you can automate the email sending process. If a test fails, you'll want to know immediately. Why wait to pull the reports manually?

Tests are for validating application logic. Frameworks are for easily creating and building tests. Following these seven steps, you'll be able to build a robust UI automation framework that will lay the foundation for enduring success.

Building Your UI Framework & TestComplete

Building an automated UI testing framework can be a painful process. It's complex and time-consuming. As the general guide book for your testing process, your framework should outline everything from the standard language you plan to write code and scripts in, to which practices you're implementing, and the tools you aim to use. When choosing a tool, you should adopt one that is flexible and comes with out-of-the-box support for a variety of languages.

TestComplete, an automated UI functional testing tool, enables teams to build and run UI tests across desktop, mobile, and web applications. It supports a wide variety of scripting languages, such as Python and JavaScript, comes with an extensive object library with over fifty thousand object properties, and has robust record-and-playback feature. With these capabilities, any team can easily build a strong UI framework.

Carson Underwood, a Quality Assurance Engineer at O'Reilly Auto Parts and an expert in test automation, built his UI testing framework from scratch. Check out his step-by-step process and watch as he walks through how he used TestComplete to accomplish each piece.

About SmartBear Software

Supporting more than five million software professionals and over 20,000 companies in 194 countries, SmartBear is the leader in software quality tools for teams. The company's products help deliver the highest quality and best performing software possible while helping teams ship code at nearly impossible velocities. With products for API testing, UI testing, code review and performance monitoring across mobile, web and desktop applications, SmartBear equips every development, testing and operations team member with the tools to ensure quality at every stage of the software cycle. For more information, visit: http://www.smartbear.com, or for the SmartBear community, go to: Facebook, Twitter, LinkedIn or Google+.