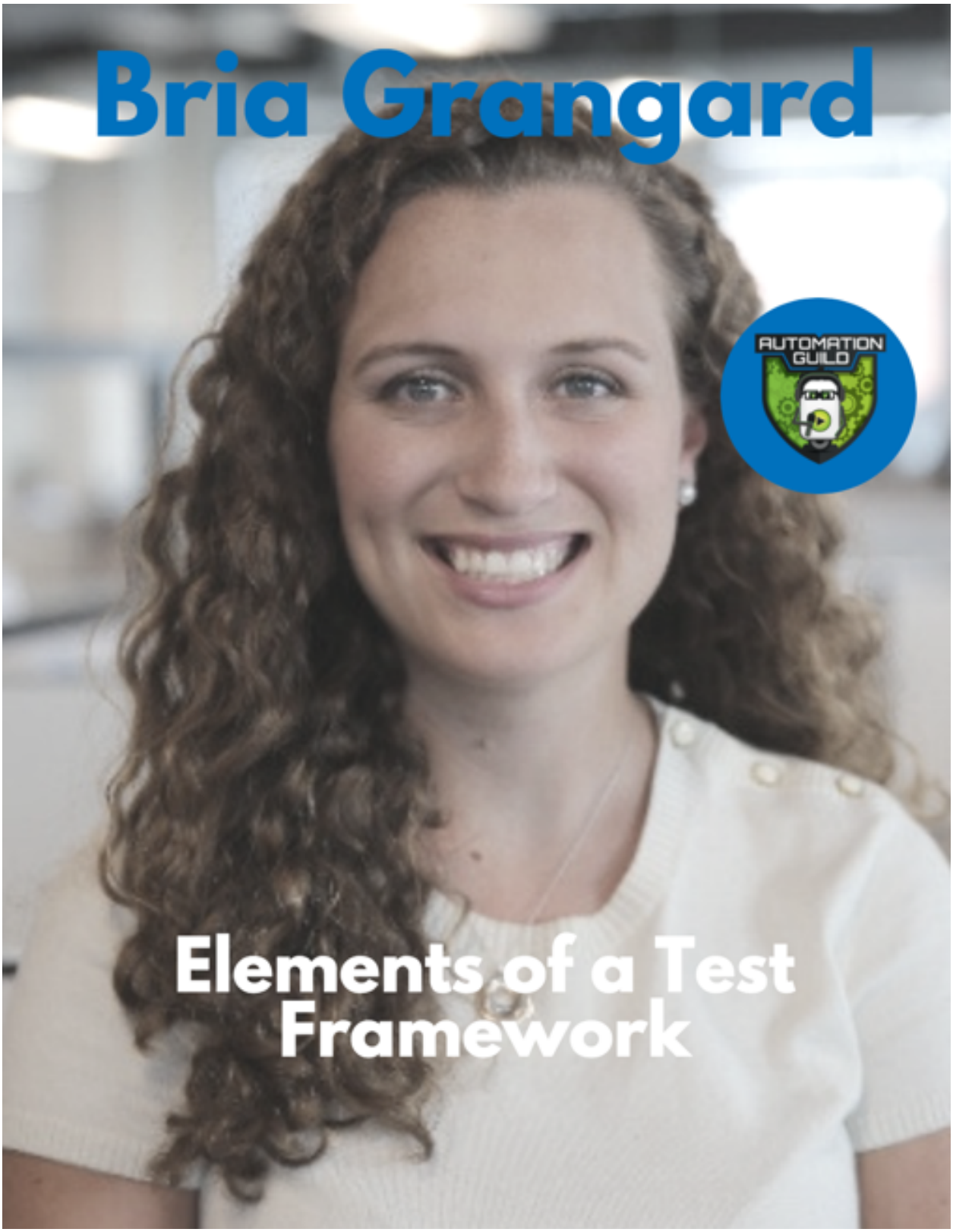
Bria Grangard
AUTOMATION GUILD
Elements of a Test Framework

Bria Grangard: Hi, everyone. Thank you so much for joining me today. My name is Bria Grangard, and I'm on the product marketing team here at SmartBear. Today, we're going to be talking all about test frameworks. How to build one. What are the steps for creating a test framework? What are the key and critical elements to making a test framework?

This was inspired by actually attending our first user conference here back in September, and hearing a lot of people say that they were struggling with this stuff. They didn't know where to begin. They didn't know what to do, or maybe they've already started testing, but they could have really benefited from creating a framework.

After discussing a lot with different experts in the field, and researching a ton on this topic, I hope that this presentation can inform you and help you in creating either your first test framework, or a test framework for enduring success. Thank you so much for listening. Let's get started.

So let's get into it. If we're going to be talking all presentation about test frameworks, it's probably important to start from the very beginning. What is a test framework? I think it's really important to start high-level. I know you all have probably heard this more times than you want to, but it all comes back to the SDLC at the end of the day.

The first step, of course, is requirements. What do we make, and how should it behave? Then, of course, we go into tests. Making sure our requirements work as stated. But within the tests, there are multiple parts. There are definitions, test sets, and test environments. Finally, we go into the defects. These defects occur when the actual results do not equal the expected results.

For the requirements, it's really important to think about: What do you want to make? What do you need to make that happen? And how should it behave? Maybe it's a form that needs to be filled out. Maybe it's a mobile game. Regardless of what type of software you're building, requirements are very vital to the success of your product. It's where both your application and framework start to get built, so it's important to spend time actually mapping it out.

This was always a reminder to me of back in the day, when we were writing out essays, and my professor, or maybe my middle school teacher, would make me plan before I wrote. It was probably one of the most frustrating phases of the writing process, but it's really important, and it can definitely set you up for success.

The second piece of the lifecycle, as I've mentioned, which are tests. I discussed how each of these are broken down into definition, sets, and environments. Finally, once you've defined your tests, and run them on different environments

you choose, those little things pop up that are inevitable at the end of the day. Defects. As I mentioned, when the actual results do not equal the expected results.

So this SDLC framework, I don't want to go over it into too much more detail, as I'm sure you all are well versed in this. But it's critical. Everything should come back to this. You should be able to list all of your changes that are tied back to each phase of the SDLC. You should encapsulate that all into a release, and everything should just be interlinked and intertwined. And a test framework can help set you up for success to do this.

So what is a test framework? A test framework, as I mentioned, links tests to other SDLC items. As we just mentioned, within tests, they can be linked to requirements. They might be linked to defects. They might be linked to different releases. While it is not necessarily an automation framework, it often contains one. A lot of times, I would get questions on a framework, and they would refer to a UI testing framework. Now while this can definitely be a type of framework, it is not necessarily a pure test framework at the end of the day.

A test framework allows for rapid creation of tests from reusable components. This whole idea of reuse is really important. You want to be able to save time, at the end of the day. Teams are always pressured to speed up their release cycle, and be able to do that faster without having more bugs. And that's something really important to be able to do. If you have a test framework with reusable components, that can definitely help you save time and increase efficiency.

Finally, it helps you separate data, once again, from logic. This whole idea of reusability, and it provides a standardized test language and reporting structure for an application under test. I think this is really important. If you are going to have multiple people who are working on a test framework. You want to speak the same language at the end of the day. It will make communication much easier, and it will help your team all be on the same page.

So how to actually build a test framework? Let's get into it. The first of the three Ds: Define. This is probably the easiest step. And you're probably very used to doing this. It all starts with defining your requirements. The very first step is to define a requirement. Each requirement should be linked to at least one test case.

I want to talk to you a little bit today about something that I was working on here at SmartBear, and something that I needed to actually test. I was in charge of launching a new website, qacomplete.com. And on that, we wanted to have an actual page that users could interact with. That was our ROI Calculator. And I'll show you a screenshot in a little bit here of that.

But there are many different types of inputs, and there are different types of outputs. There's a journey I wanted to walk my user through. I wanted them to

be able to insert an input, have an output, and then be able to either share that output, or take it in and acknowledge that it's the output they like or don't like, and exit the application.

So it's really important to think about the workflow, and the journey that you want your user to go on. If you're already using your user stories, which many people are today, you're already mostly there. But it's really important to think about why. Why is your user going to go through this? Are they really enjoying playing a game? Are they going to go check their most recent banking transactions? What is the common workflow that they're going to go through, and why are they going to go through it?

This part of defining is really important. You need to ask these key questions: Who is using my application? Why are they using it? And what context are they going to be using this application in? As I spoke about a little bit, the QAComplete ROI Calculator. I had to think about who was going to be using it. A QAComplete prospect, hopefully, right? The perspective: They're interested in determining what QAComplete can do for them. And finally, putting it all in context on the ROI Calculator page, in a web browser, on qacomplete.com.

Finally, another important step of defining is figuring out the desired path with minimal variation. Well, this gets way harder as you scale up the complexity of your applications. It's very easy for me to talk about this, and say it's very easy to do. I tried to give you an example, to show you how the easiest examples can still result in a ton of complication. But it's really important to think about what is the minimal variation? For out example, we have four different inputs, or leaves that were defaulted. They then, the user would receive three pieces of returned data. The user could click Print, or the user could click Contact Us.

We had to look at this path, right? And we had to figure out what were the steps that they were going to go through, if they left all the default values, and just clicked, for returning the data, if they just clicked the Print with all the defaults set already? And how could we minimize the amount of variation at every step of the workflow?

That is defining. You need to make sure that you're looking for the key journey that your user is going to experience, and that you want to take your user on. You need to define every phase of this. The nitty-gritty. Get granular. You need your inputs. You need your outputs, and you need your exit strategy. And within all of these things, you're going to have 10, 20, 100 different options as well.

Now we'll move on to the next phase of how to build the test framework. The second D: Decompose. The next step in building your testing framework is decomposing your workflow into the smallest, most atomic pieces, and separate actions, associated with each step. This is important. Keep in mind that it's not referring to unit or unit testing. Unit testing is designed to test specific, or niche, functions, to ensure they behave per your requirements. However, in this state,

the decompose stage, we're looking at it from the user perspective, and not taking the backstage actions that are triggered into consideration.

It's important, we have to step back. We just asked ourselves "Why is the user going to be using this?" And now we need to, once again, decompose from that user perspective. You'll break down your workflow into a similar manner. For example, actions that provided input, data, results of those actions, the workflow logic, and exit points from the workflow.

Here we are going to take a look at this ROI Calculator. Here you can see I've outlined the different inputs. There are four different ones. The user can select or type in values for the number of team members. The user can select or type in values for the number of requirements. The user can select or type in values for test cases per requirement. And finally, the user can select or switch between SAAS and On-Prem.

Here it's important to note, there are multiple ways they can input, right? For the number of team inputs, they can use the sliding method, or they can type into the box here. That definitely adds variation. That definitely adds complication. Then we look at our different outputs. Coming over here, we can see that there's an output up here. The implementation cost. There's an output right here, of the savings. And finally, there's an output in the ROI.

Then you have to give your user an exit. A way out. In our case, we gave that to them in two ways. The first is through the Contact Us button. The second is through the Print Results button. Getting to this level of granularity is not necessarily fun, but it does require an in-depth knowledge of the application under test, and it is absolutely vital to the decomposing stage of building your framework.

We figured out what the inputs are going to be, and then we got so granular as how they were going to actually input these inputs. Same with the outputs and the exit strategies. We really touch on the first two phases of how to build the test framework, of the two of the three Ds. Define and Decompose. And now we're going to go into probably the most difficult part of building your test framework, which is actually the third D, in the Deciding phase.

For the three Ds: Decide. The third one. The most important! Just kidding, they're all important, but definitely the one where you have to use your gut and make some calls, or think about your user's journey. Or really think about your team, and what can be automated, and what can't? So on and so forth. Now it's time to make some decisions.

The third stage in building your framework is decision-making. This will be tricky. There are a few choices you'll need to make. For example, maybe what environments are going to be used, for different operating systems, browsers, devices. Some other things that might be used. Security permissions. Who can

actually access the different tests that is going on. The user roles. Who's going to be going through it? And finally, conflicting software. This is an interesting one that I'm looking forward to touching on in a little bit.

So if you're decomposing all your major workflows, it's important to find your commonly reused steps and actions. You can find actions and checks that use the same data. For example, when you saw the ROI Calculator, you saw that there are two different workflows. Workflow One, and Workflow Two. Workflow One was focused on inputting your data, and Workflow Two was focused on using the slider. These were two ways of inputting the same data, but two different ways of doing that. It's really important to find actions and checks that they use the same data for the same outputs, but you can do it differently. So you've got to break that down.

Now you have to decide on, "Well what are going to be the most commonly used steps? When should we be using automated testing?" These are the important decisions that need to be made. As I mentioned, a decision that's going to need to be made, automation. What do you automate? A question I get over and over again, and let me tell you, the answer is never simple. From environments, through data entry, backend data, repetitive and boring tasks, tasks with high reuse, tests with timing, non-functional test types, and capturing results. These are all things that you can automate.

For example, repetitive and boring tasks. These are so prone to inattention errors. I think if your job was to sit there and type a one all day, you'd be likely to maybe slip up and type a two, or a squiggly line next to the one. You never know. The tilde. It's really one of these things that you could automate, and not only would it save you time, but also it would most likely increase your accuracy.

This is a list of things that you can definitely look at automating at the end of the day. Choosing what to automate is complex, and there are very few tasks or actions that have this clear cut answer. Hopefully this list can help you in making those decisions.

Another thing I'd like to highlight are the non-functional testing types, such as performance and load testing. These can be great candidates for automation as well. These can be used to estimate how an application would handle real user scenarios, and emulate bandwidth. These are vital to understanding where the breakpoints of an application lie. Fixing performance and load issues takes a lot of time, and can require additional fixes beyond just code changes. Automating some of that can be very helpful.

Finally, report generation, or capturing results. If you can figure out a successful way in automating this, you would potentially eliminate the need to manually generate reports that capture, consolidate, display on a dashboard, the different data that you're using, and the analysis that you would like to do. You could potentially generate reports monthly, weekly, and even daily. And doing

that on a regular cadence can be a big hassle, and automating it could save you a lot of time, and definitely a ton of headache.

Finally, I want to discuss what not to automate. While automation can definitely speed up things, there are definitely things you cannot automate, or things that might be more of a hassle than what they're worth when it comes to automation. It can speed up testing cycles and improve coverage, but as I mentioned, not everything should be automated.

A question that pops up frequently is "Why not?" The answer is there are actions that are better left to a manual process, and as long as humans are involved, there is always going to be a need for manual testing. Some examples might be: Do the colors on your webpage clash? Is the text hard to read? Is the workflow user-friendly? As I mentioned, as long as humans are interacting with machines, and with the applications you're developing, you'll need to conduct some form of manual testing to find defects like these.

We've discussed the three steps to create a successful test framework, but I want to highlight just the elements of a test framework. I'm getting a little short on time, so I'm just going to once again show you the SDLC, but not walk through it in as much detail. But now I want to show you the list of the elements of a test framework, and break them down into a little bit more detail. The list I'm going to be going over are libraries, test data sources, helper functions, test environments, modules, and structure or hierarchies.

So let's start out. A library. This is the first part of your framework. This is a repository of all your decomposed scripts, and they should be separated into their components, as you'll need to start with parts that are easy to trace and easy to build into your test framework.

The second one are test data sources. Not only will you want to separate each of your decomposed scripts into their individual components, you'll want to separate your scripts from the data. By separating your test script's logic from your data, you will ensure your scripts are reusable and easy to maintain. With data that is hard coded, you're forced to rewrite the scripts when changes are made. Otherwise, the test will fail. An example of this could be inputs and expected results used by both workflows in one data source.

Next are helper functions. Combining your test environments with your helper functions will be a key to your success, so we're going to review those both now. Helper functions are scripts such as setup scripts, cleanup scripts that are used to scrub the database, maybe scripts that you use to access external information, etc. These are the catchall for anything that isn't a user action.

Helper scripts are awesome. For example, you'll want to write a setup script for a Mac device that's working with Safari, as well as a script for a Windows machine that works with Internet Explorer. Helper functions are absolutely

necessary to complete the other tasks that aren't associated with a particular user journey.

And now test environments. We've talked about this, and test environments are only going to continue to get complicated, as who knows what Apple is going to release for their next device in the next month, and what operating system will be used. Hopefully, we keep things on iOS, but you never know, and we continue to see in this whole world of the internet of things, just explosions in what is going to be out there. So test environments are only going to continue to be crucial.

You'll want to define the list of environments you'll want to cover. Who is accessing your device, and from where? Are they accessing from all over the world? Is it North America specific? Or is users of only a Windows machine, or are you also including users of a Macintosh as well? Or maybe others as well. What different browsers are they going to be using, for example? Are they going to be using Safari, or are they going to be using Edge, or are they going to be using Internet Explorer, and how far back in the versioning do you want to cover?

The number of devices and systems available, as I mentioned, it's only going to grow. It's crucial to take these groupings into consideration when you're considering where your application is going to be accessed, and what you want to make sure you cover at the end of the day.

Finishing up here on the elements of the test framework, next up is modules. Once you finalize the first four pieces that we just went over, you're ready to start building the bigger pieces. The modules, we call them. Each module is a combination of a library item with their helper functions, environments, and data sources, that together can highlight one single product capability. So modules are laid out this way, and they will enable you to link the defects more easily back to specific requirements and tests, and hopefully tie that all into a different release.

Using the ROI Calculator once again. Workflow One. Input: The ROI slider. Using Data Source A. Then you have the ROI Calculator checks using Data Source A. Then you add the Workflow Two input: ROI type in. Actually typing in what number you wanted, using Data Source A. The ROI Calculator checks using Data Source A, and then repeated for coverage of environments for Internet Explorer, Chrome, Firefox, etc., wherever people are accessing those from.

This is so important, and I'll give you an actual example why. If you go on this website today, and you look, we have a bug in it. If you use the slider, you can only go down to a minimal number. I believe in the website that it's set as one. However, if you type it, you're able to actually type in the number zero. These values both change the outcome values, and so clearly, there's a bug. Either we should set the minimal typing value at one, or we should allow the slider to go all the way down to zero, even though it might not make the most sense.

It's really important to break down these modules, and say, "Okay, if I tested all of this, and they're linking together, and I'm having one issue with one workflow and not with the other workflow, or I've noticed something is different, what module is this tied to? What bug is this tied to? What test is this tied to? What requirement is this tied to?"

Finally, we are going to cover structures and hierarchies. This is interesting, and I like to think of it as, when I was in middle school, and I got binders, and I loved to use the pocket dividers or folders to organize and save homework. Tests and quizzes. Assignments. So on and so forth. To finalize, you'll want to develop a folder structure for your modules in a similar manner in which you utilize dividers in a binder.

These can be sub-categories. These can be however you think will work best. Each folder represents the parent and child relationships you can use, and your overarching structure. I have found, and I have heard that the best and most successful way to do this, is it should represent the format of the application under test.

If you are going down a typical workflow, and you're guiding your user down a typical workflow, then you should be able to see that this requirement is tied to this workflow, tied to this test, and have it all documented in a parent-child-like relationship. This will help set you up for success in the long run. Using a structured hierarchy format that you can use and make sense to not only you, but also to your team, can be very helpful at the end of the day.

We've talked about all of these. You have defects. You have requirements. And you have releases. These all tie back to the SDLC and are important to acknowledge that your framework, it links back and addresses each of these components.

So with that, I want to say a quick thank you. Thank you so much again for joining me, and of course, if you have any questions, feel free to join me for the live Q&A session. I look forward to seeing you all there! Bye.