Richard Bradshaw - Don't be SCARED of automated checks/tests

RichardBradshaw: Hello and welcome to this Automation Guild 2018 talk. My name is Richard Bradshaw, also known as the Friendly Tester. You were probably tuning in to expect a talk titled Don't Be SCARED of Automated Checks. When I submitted this talk to Joe, that was the title and SCARED was the acronym I was going for what I'm going to talk about today. However, upon reflection I realized that SACRED might fit better. The talk's exactly the same. The description still matches, however, we're going to go with the title of Your Automated Checks are SACRED. SACRED fits a little better for me. We shouldn't be scared of automation. I'm not trying to put people off. I'm actually trying to encourage more people into it, and SACRED just seems to ... Just fits. We want to treat our checks with respect and they're incredibly important to the team, and especially going forward a lot of teams are already getting immense value out of their automated checks, and I hope more teams going forward do.

Also, the order fits it, which you'll see through the rest of this talk. SACRED was the result of me spending some time thinking about basically anatomy of an automated check. What's actually the make up of a check? When we go through and we sit down in an IDE and we start coding, we are doing far more than we realize, and I wanted to break down and try and understand what these were. The result of that was these six letters. Then once I played around with some words, as I said, I ended up with SCARED and then I changed it now to SACRED. The talk, we're going to go through these six letters. We're going to look at what they mean. We're going to look at who should be involved. We're going to look at when you should be involved, and we're going to look at how you should go about doing them, and why it's important to consider these things in our automated checks.

What I'm hoping you'll get out of this is a realization for the skills that are actually involved in creating really fantastic automated checks. We're very fixated on code as an industry, and I feel that we need to focus a lot more on some of these other skills. We're pushing some very good testers into learning how to code, and potentially as Dot Graham's quote that she stole from someone else, I can't remember who, if we force good testers to code, we'll gain a bad programmer. What I want to try and demonstrate is that there's more to automated checks, there's more to writing those checks, creating those checks, thinking about them than knowing how to code. I think we're in danger as an industry of focusing on code too much.

Also, other members of the team have lots to contribute to some of these areas, as well as the testers who don't wish to code, how can they still contribute to automation efforts? They really can. They really, really can. I'm going to break this down, and also I'm also hoping it will realize for those who have been writing automation extensively for a long time, get a realization for the bits that they may not be aware of they're doing and now they have a way of tuning in

and focusing on that so they can improve overall. Let's get into it. Let's jump straight into S.

Just like magic, we're back, and we've now got a new slide. Using the whiteboard, as you can tell, is kind of in homage to my YouTube channel, Whiteboard Testing. I love the whiteboard. It's more visual. I tried sitting at my computer and talking to it and using PowerPoint and something. Didn't really work for me, so we're going with the whiteboard approach. The S, the S stands for state. Why is the state important? Well, if you think about an automated check, we're doing some behavior on the UI or perhaps we're hitting an API or even if it was at a unit level, we're calling some kind of object, and therefore we need to have the state in place. For example, most of our applications are driven by data. It's data that makes the UIs appear as they are. It's data that makes the APIs return what they do.

If we can't put that data in place, therefore we're not going to be able to control our checks as well, so we have to think about state. In order to think about state, we have to be able to understand how our feature works, everything that goes with it. What data [inaudible 00:04:11] in the database, what tables are used? Does it call third-party APIs? Where does it get all this data from that makes the UI look like it does, or makes that API return this JSON, because it's that that we then do our assertions on later on. We need to be able to control it. If we can control it, we can then make our checks more deterministic, which means we'll always have the same results every time, so therefore if we want to always have a 100 users when we do our automated checks, we can put 100 users there.

If we need a user who lives at 123 Test Street, we can make one. If we want to see what the system behaves like with no users, we can make that happen if we put all the code in place to be able to control that state. As I said, it's the data that drives the interface, which is why we have to consider it and we have to be able to control it. A most common term for this would be testability, and it's very easy to think about. We ask ourselves how easy is this feature to test? I like to do this in pre-planning, I like to ask the team, "How are we going to test this feature? How am I going to test this?" Think about it first, we always have to do this stuff manually, and then we can automate it. If we can build something upfront to help us test it initially, when we come to automate it, it should be significantly easier.

The when for me is always at feature design is when I want to bring this up. I want to do it before the developers have written any code, because the likelihood is that my requests or the ... Well, not necessarily my requests but the team's requests to be able to have some automated checks in place means that the way the feature's designed may have to be changed. Perhaps they might add in an API for me that I can use to create some data, or perhaps they might design the tables slightly different to make them more automatable. I don't know. If you don't bring this question up at the start, in my experience, going in afterwards and trying to add in testability is ... It's significantly harder. In some

cases it will mean a lot of rework, which the product's owner or the team might not have the capacity to do. There's lots of features to get out the door and therefore it gets put on the backlog or it never gets achieved.

Whereas if you bring it up at the start, it can be designed into the feature, and therefore you'll have it from the beginning. Some ways we can look at testability. Firstly we need the skill of modeling. We need to be able to think about the feature, think about where it gets its data from, how can we put it there, how is that data used, and how is that data changed as you go through the flow of making API calls or going through a process on a UI. We need to understand those data changes so therefore we can control them and we can manipulate them, we can model them, and we can build stuff that helps us change that.

One of the things you can build is called the data builder pattern. It's a very simple pattern. You have a model of an object. You then have something called a builder where you can have predefined ones or you can pass in values. For example, a user where we might have build a new user, which sets all the flags to be all as if it was a brand new user in our system, or I could have us build a user that's been with us for five years and it will add in lots of data, set their creation date in the past and behave that way. You can have builders for anything. Then the final piece of the data builder pattern is something called a creator, so you have a piece of code that is able to create that object actually in the database, actually in your environment. That could be an API call, it could be a SQL call, it could be a Mongo injection. It could be anything, but basically you have a creator.

This is a really good way to control your data. You do this at the beginning of your check. The data builder pattern relies on having some hook. You need a hook into your system to create the data.

One that's becoming increasingly more popular is something using mocks and fakes or stubs. I don't really go into the huge difference of them. It's not for this talk, but basically instead of talking to the real database we put something in place that we build. We can build that. A lot of automation engineers out there, we're using common tools. We might be adept in Selenium, or we might be using some other JavaScript tools in the UI, and we're very familiar with things like REST Assured and all these libraries out there. If you actually take a step back, you've probably got a lot more coding skills than you realize, and you can easily apply those to be able to build something like a fake or a stub.

I personally built one for a mobile app [inaudible 00:08:34] been testing a few years ago. Instead of hitting the backend, which was quite difficult to control; it had very poor testability, I was able to build my own API using Node very quickly to get up and running, and I pointed my app to that and therefore I could control the behavior of my UI and my app, and therefore made it more automated, more and more testable. You can also build hooks, so a common example of this is the user interface. If you're doing some UI automation, you'd

probably have to authenticate, you have to log in every time you want to do an automated check if you're on some kind of secure system. That is a slow process, and also it adds a blocker into your checks.

If that log in process fails, none of your checks will run. Therefore, you're waiting. You're going to delay that feedback loop, which is not good. Can the developers build you in some sort of backdoor? Again, don't get me wrong, that comes with other risks as well, but thinking about these hooks, thinking about how can you get the checks right to that point where you can do what you're interested in. If you've got a check that checks the profile page, you're not interested in logging in, but now logging in is part of your check. How can we get straight to that profile page, put the data in place immediately that's there, bypass whatever we have to do so we can focus on the thing we're trying to test, make them more targeted, straight to that point, straight to that page, straight to that API call?

For example, in the API space do you have to authenticate every time or can you bypass it and just pass in a token that always works? Something like that. We have to think about them. We have to design them. Everyone can get involved with that. This is not an automation engineer's sole responsibility. Product owner needs to be aware that you want testability, because then they'll design that into their stories. The developers can help you. She could build you an API that gets in there. She could build you queries ready for you to be able to automate with. Then the testers who have been using the product. If you are an automation engineer and you're considered separate to the test team, the tester will have loads of knowledge about this feature, about the data behind it, and they probably already have models in place from when they were doing their test design that you can take advantage of. Everyone can help you do this.

Some questions to ask yourselves about state that you might not be thinking about. Go and have a look at some of your checks and the features that they're using. Where does it get its data from? Do you have full control over it? Are you relying on something else? I've seen some really awful test suites where they're relying on the previous check, so previous check leaves data in a certain state for the next one to go, and that's bad because now you can't run them in any order. In an ideal world, you would run all 100, 1,000 of your checks in parallel. They should not be dependent on each other and controlling the state allows that. Can you create any data object you need in your database or do you have access to it? Are you relying on an API that's perhaps not designed to do what you're doing, or are you going through the UI to create some of this data? Have a think about that.

Then think about who can help you with test testability. If you spot an improvement for testability, tell the team, educate them, try and get some buy-in. Then ask yourselves every time how testable is this feature. You've got to remember that we're talking about automation here, and if you can't do something easy enough manually in terms of exercising an API or some interface, it's going to be significantly harder to do it with an automation tool.

Ask yourselves these questions. The first one is state. Very important. It's the start of the process. You control the state, the rest of the stuff will follow and be deterministic. If you can't control the state, the risk of flakiness, the risk of brittle checks is going to be very apparent. Think about the state.

Okay. Welcome back. Board's updated. Magic. The next one's A, and A stands for algorithm, so what I'm talking about here is basically the flow that you go through to get to the assertion. We've set our state, our state's in place, and now we're going to have to exercise our system, we're going to have to exercise the behavior, and we're going to have to do that in a certain order. It could be making one or two API calls. It could be clicking through several screens on the UI. That is our algorithm. It's basically the make up of the check, what we're trying to get to until the point of our assertion.

Why it needs special attention. A well-designed check will have a very clear purpose, and it will be highly maintainable. If it's highly maintainable and it's very clear what the actual flow of this check is, it will increase the speed of designing it as well as the speed of maintaining it going forward. Now, in terms of when we do this, I've put post testing, so the reason why I've said this is often ... It could be a bit of a misconception, but I have personally met some automation engineers who claim they don't do any testing. I'm just an automation engineer. I just write code. Now, why I say the algorithm is post testing is it's incredibly rare that you will just solely sit down, don't look at the system at all and create a check.

You probably go back and refresh your knowledge. You might click on the UI again. You might wonder is that JSON right? Maybe just make an API call just to check. It's very rare that you would just sit down and run it. Even if you do just sit down and type it and type it all out, the first time you run it you're watching it very closely, like, "Oh, please work. Please work. I think I've got this right." Then you're actually testing there. You're just using the check as your tool to get to the point where you're happy and you think, "Right, [inaudible 00:14:31]." Your check is now fit for purpose. We do it post testing where our knowledge of the system is clear. We understand the flow. We understand the behavior and we understand everything that we need to put into our check.

Now, for example, things like waits on the UI, they're incredibly important and you wouldn't be able to do them and get them right without exercising the system, therefore without testing the system, but you're testing the system with a focus about check design. You're not testing it to check the feature's correct. You're testing it to see how it behaves from an automation ability point of view. We're looking for waits, looking for indicators, looking for things that might get in the way of your check. That's when you're doing that different type of testing. Therefore, we'll go in, we know about the system. We've done some actual testing on the system. You understand the behavior, and now we're doing testing to discover how automatable it is to make sure we can get that flow right and give our check everything it needs.

You've got to remember, these checks are stupid. They don't know anything. They're dumb. We have to educate it, so in order for us to educate it we need to make sure we know as much as we can about that behavior, about the make up of that API or that UI, and therefore we can design our algorithm appropriately to make sure it can handle all the situations we're aware of. That's how we do it. It's good test design, for example, so if you do good test design, you'll get a lot of information and you'll be able to push that into your main check. System knowledge. We need a lot of knowledge about the system to be able to get this right. As I mentioned, behavior knowledge as well as what's going on under the hood. How is our system behaving? What could happen? Is there a flow on a screen where another window could appear? Do we have to deal with that, or can we control the state as we spoke about previously?If we can't, we have to improve our algorithm to be able to deal with such scenarios.

Naming and code craft. Once you've writing this algorithm, we've got to be able to maintain it. Our systems never stand still. If the system stood still, you wouldn't have a job, so the systems are always moving, therefore we have to be able to maintain these checks quickly, and if we get our algorithm well writing, good code craft, good standards, it should be easier for us to update and maintain it. Again, everyone can help, because as I mentioned you need system knowledge. Product owners, BAs, testers, they have this system knowledge. We need to know what's going on under the hood. The developers are the perfect people to know exactly what's going on under the hood. The automation engineers need to have the skills to go under the hood, go look at the make up of it to make sure that they're very aware of everything that could happen. We need to pull in all that knowledge.

This is the key point, this is the algorithm. This is taking us to our assertion. It has to be right otherwise we're going to introduce brittleness. If we don't get the waits correct, if we don't have the right different flows, if we're not aware of all the flows, we're going to introduce flakiness and brittleness into our automated checks and we don't want that. We want them to be smooth, deterministic, and do the same thing every single time. We need to make sure our algorithm is rock solid. You've got to give this a lot of thought. This is difficult.

Now, a lot of the time we do this in real-time. We're aware of a feature, so we go on to it. We look at the screen. We look under the hood, we write some code, we test it, we go back. We write a bit more and we run it, therefore we're doing all these things in parallel, but sometimes we can be a bit distracted and just say, "I'm just coding. I'm just writing my check." You're not. You're doing a lot more and we need to understand that so we can improve on the different areas.

Some questions that you can ask. Go and have a look at some of your existing automated checks. Are they starting at the right place? Are they actually starting in a sensible place for what it is you're trying to check? We mentioned this in the state. Are they starting on two or three pages earlier than they

should be? If they are, let's change that. Let's get our algorithms shorter and more concise and more solid. Is the intention of the check clear? Have we used the right terminology or the right words so that we understand what this check is trying to do? Can someone else in your team come and read your code and know exactly what that check is doing? If they can't, then perhaps we need to think about our wording. Is your wording concise and consistent with the rest of your application? Got to think about this stuff, because other people than you may end up reading some of this code, and they need to be able to understand quickly.

Then finally, is this even the right layer? As you're going about designing your algorithm, you may go, "Why am I doing this on the UI?" If I'm just caring about these four values, I know they come from the API because I've been studying the state, so why don't we just do this check on the API? This is all JavaScript. Why don't you do a JavaScript test? Why are we doing it on the UI? I'm actually trying to look at something visually. Why am I doing it on the UI? Why am I doing it with Selenium when I could do it with a visual tool? Again, some great questions to ask. A is for algorithm. How do we go from the beginning of our check once the state's in place, run through to the point where I'm ready to do my assertion? Algorithm. We've got to think very clearly about that.

We're on to C. C stands for codified oracles. If you're not familiar with the word oracle, it's essentially a way of how we decide if something is right or wrong, or how we decide if something's worth more investigating. It tends to be is there a problem here. A common example I like to think about would be if you went to a door and the door handle wasn't where you expected it to be. If the door handle was four foot high, you would probably say actually is there a problem here? Because you've got an oracle in your head that door handles should always be around waist height, for example. Therefore, you would question it and you'd be like, "That's different." Then you would go and uncover information.

As a test can any of you spot what's wrong with this slide, this drawing? I'll give you five seconds to think about it. Some of you will be screaming at your screen now going, "The C isn't red. The C isn't red." Yeah. The C isn't red. Two letters ago you wouldn't have had that oracle, but you do have that oracle now because you've spent the last 10 minutes watching this talk, so therefore you've now got this oracle in your head that Richard's going to make all the letters that we're talking about red, and therefore I didn't in this case, so you were like, "Perhaps there's a problem here. Perhaps he's made a mistake, or he's done it intentionally to try and teach you about oracles." It's definitely the latter.

When do we do this? We do this while testing and exploring. As we're testing and exploring, we're looking for problems, we're looking to see how the behavior of the features are and if they're correct or not. How we determine if they're correct or not tends to be by some form of codified oracle. In most of the instances it could be a story in JIRA or you might have a tracker you're using that says, "When I click buy now, I should get a window that says processing and

then I get system confirmation." A lot of the time they're very vague, and what we do as testers is we add all our own oracles to it. For example, it could say if you don't fill in your credit card details and you click pay, an error should be displayed in red that says, "Your payment, you've not filled in your credit card details."

When we come to test that, all we're looking for is to see if we get the error and it's red, but what you actually probably did was you checked that it was red, you checked the font was readable, you checked the text size was readable, you checked it was in the right place on the screen. You checked that no other errors appeared on the screen. You checked that the whole UI didn't reorganize as it appeared on the screen. That's what you actually did, whereas if you told an automated check just to check that the text was on the screen and that the class was red, that text could be anywhere and it would still pass, so we need to get better at understanding the oracles that we use so we can educate our checks on which ones it should be using.

If we're not very good at that, we're going to end up with checks that will always either always pass or the oracles are very weak and therefore we could end up releasing some bugs into production that we thought would never happen because we had checks in place, but it turns out our oracles weren't good enough. Our state was correct. Our algorithm was correct, but we failed on our oracle, and therefore we need to give them some more thought. The way we can do that is with our knowledge. You're always gaining knowledge of the system continuously. It's really important to think about this continuously. The system is always on the move. It's always changing, and therefore our knowledge is always changing, and we have to go back and reeducate the system, reeducate the checks to bring them up to date with our current knowledge.

You've probably got some oracles, some assertions in your current code base that are pretty weak, and if you went back and found them, you would certainly add to them. That's because at the time you mightn't have been as good at identifying oracles as perhaps you are now, or perhaps after this talk you'll go and do some research into oracles. The rapid software testing namespace has some fantastic materials on oracles, and there's also a really ... It's quite an old piece, but a really good piece from a chap called Doug Hoffman, which has a lot about oracles as well.

We also have to reflect. We have to reflect every now and again and go, "How did I know that was correct? What was I doing? What was I thinking?" And trying to put that into our code, otherwise we're going to have these poor assertions. Product owners, BAs, testers, all are going to be fantastic in helping us do this. Product owners especially, because you've probably all had the same experience as me where you get a feature, you read the acceptance criteria, or you read the expected results. You do some testing. It's met everything that's in the ticket. You demo it to the product owner [inaudible 00:25:01], "Nah. That's not right. That's not right."

The reason why it's not right is because they failed to actually come and put across what they wanted to see, and therefore they're very good at now being able to help us identify whether our oracles were good enough in the first place. Those would be our testing oracles, but after that those are the ones that we try and put into code. The difference with why I've called this a codified oracle is because as soon as you codify it and that basically means put it into words, put it into code, make it permanent. Because obviously code can't change without us at the moment. Lots of talk about AI, but some people haven't even been able to write any automated checks yet, so we'll park that bus for a while.

We need to be able to educate the code, because it can't do it itself. Once they're codified, they're stuck, and some of them will still be strong and still perfectly valid, but other ones could change. Perhaps they're a bit weak now. Perhaps they should be looking at two or three elements on the screen. Perhaps instead of just checking the one JSON value, it could be turned into a contract type test that checks all the JSON. Who knows, but we have to continuously review these. Codified oracles, very important. Basically your assertions. You're probably thinking when you're reading this, you should be thinking about assertions. We need to get these assertions solid. We need to be really solid to make sure that they are asking the right questions of our application, because we're not there to ask them.

We've written these automated checks to do that for us, because we don't want to do it, but we've told it to ask very specific questions and we've told it what the answers should be and if they're not strong enough, stuff's going to get through. Sometimes they might be too strong and they'll fail for different reasons. They'll fail at something that actually isn't really part of the check at all. We would have been happy to ship it, but for some reason now we've made our check too strong. That's another issue we have to be aware of. We've got to continuously review these. Some questions you can go and ask yourself. How strong are your oracles? How strong are your assertions? Have you got any of the assert true is true, which people put on Twitter all the time.

Have you ever seen any of your checks fail? Sorry, not any of them. Have you seen that check fail? One of the things I love to do personally when I'm writing a new check is make it fail. Before I commit it, I want to see it fail. I want to know if I've written a good oracle. I want to know if it's possible that it could fail. Therefore [inaudible 00:27:31] want to exercise it. I tend to do it a few times just to make sure that it is a bit solid. There's actually a post on my blog about who tests the checks. Have a read of that. There's some great advice about how to make sure that you've got something solid before you stick it into the repo.

How do you know the behavior is right? You're writing an automated check. Perhaps you've been tasked with doing that, or perhaps you've done some analysis or risk analysis and you've gone, "This is an important automated check to have." How do you know that what you've decided is right is right? If you're not entirely sure or you're not aware of why you know it is right and you don't tell the code that, again, you're going to end up with poor assertions. The last

one, when was the last time you went through and reeducated some of these checks? As I said, screens change, layouts change. You might have had an assertion there at the time just checking that the text was okay was enough, but perhaps some CSS changes have gone in recently and now it changes color or perhaps it looks slightly different. Perhaps there's a little icon. Are you now looking at that as well, and if you're not do you need to reeducate your code?

I've also just written an interesting piece on the illusions of green, which is another interesting angle to this codified oracle. Just because the check is green doesn't mean everything's okay. It only means that the oracles you've coded have been met, so therefore you need to be very aware of them and make sure they're as strong as they can be. To do that we need to involve people who have knowledge about the system to make sure that you've put in enough to be able to say, "Yeah. This is good enough. This is addressing the risk that I set out to achieve." That's what the C stands for, codified oracles.

We're on to R. The R stands for reporting in the SACRED model. We're taking two angles to reporting here. The first one is reporting the actual outcome of the check, pass/fail, and any information that goes with that. The second one is actually reporting on the check itself. Is it healthy? What did it do? Does it collect enough evidence? These the two angles I'm trying to think about when we come to think about reporting. Why do we need to consider this? Firstly, visibility. We write checks to help the team make decisions. We help them with checking that our knowledge and the knowledge that we think we have is still valid or as it changes or is there something we should be aware of.

We use it to get fast feedback, so what is the visibility? What visibility do we have of these results? Are we simply looking at a build radiator that goes red or green, or do we have more in-depth analysis that we can look at? Can we look at the trends? Is there any patterns? Is there a check that continuously is going red, green, red, green, red green, and if there is can we investigate why that may be? Is there one that seems to fail on every second Monday there's one of these checks will fail. Why is that? Do we have that insight? If we don't, perhaps we need to build something that can help us do that. The other one is evidence. A lot of automated checks, they do a lot of stuff and we use them to help us make decisions about releasing and the quality of the product, but are they also collecting any evidence?

Perhaps you're in a regulated environment where you need to prove that testing took place. Are your automated checks outputting enough evidence to help you with proving that you did this testing? Are they taking screenshots. Are they getting log files? Are they writing out the steps that they did and when they did them? The kind of evidence that we apparently get from test cases, but that's other topic. Our investigation speed. When we have failing checks, how quickly can we get in there and actually find out why it failed? A lot of the time what we tend to do is whether we downloaded the latest code base, we run the check that failed locally and we watch it, waiting for it, waiting for it. I know why it failed, and then we go in and we fix it and we try again.

Can we put some logging events into our actual checks so that soon alls we have to do is opening our CI tools, look at the log and we immediately know why it may have failed, and we can start digging that way. Can we give ourselves a headstart is what I'm trying to think about with our automated check and with our architecture. That's how we do that. We do it with our architecture design. I don't like to talk about frameworks and things like that. For me a framework is something we downloaded from the internet, a library. J unit and N unit are examples of test frameworks. Selenium's an example of a UI automation framework. I like to talk about architecture.

When I build automation and when I create an architecture for my automation, I like to add these things in there, and I consider it part of my architecture. I'll have lots of libraries and features that are purposely designed to help me with the two types of reporting that we've just spoken about. Also, when we come to design the check itself, if it's something that's a bit, potentially be a bit flakey, we're aware of a few issues in that, I'm going to add in a lot more logging around the areas. I want to know. I want to find out this information. For an example of this, less of people in WebDriver for when they're doing WebDriver waits, they may set that wait to be something like 60 seconds, 30 seconds, 20 seconds. Every time your check runs you have no idea how long it's getting to.

What if it was getting to 20 seconds at every single time? Your check would still go green, but as a user if I'm having to wait 20 seconds for everything I'm doing, I'm going to not be very happy. You're not going to know about that, because your check's green and you haven't gone looking into it. Thinking about something like that, do I want to add in some output to my waits? How long did I actually wait and can I graph that? Can I stick that into a database perhaps, and therefore I can look at my average Selenium wait time and see if my checks are in good health. I'm not talking about adding performance into our automated checks. We've got performance testing to do that kind of behavior, but I want to make sure, I want to find out if my checks are smooth, whether my algorithm's flowing straight through or is it having these little hiccups that I could perhaps iron out if I knew they existed. I would only know they existed if I had some sort of reporting.

The way we can do this, we can add listeners. There's lots of listeners in pretty much all the libraries that you'll be using and you can take advantage of those. You can make yourselves some abstraction layers on top of those to report how whenever anything is happening. For example, logging in WebDriver, you can use the event listeners and just output everything that happens. As soon as there's another click, you can have something that says, "I clicked element A." Then when you have a WebDriver wait, you can say, "I waited 15 seconds." You can build all that stuff relatively quickly, because the opensource community have built these features for us into those tools. Then also [inaudible 00:34:50] exist in every other tool, but we can add that in and it's important because it will help us when it comes to this investigation time.

We can build dashboards. Instead of just a red/green, we can build a dashboard to help us look at the actual health of our automated checks. Are they healthy? How long is the execution time for certain checks? Is it going up and down? Can we then drill into that and find out why it's going up and down? We can add in analytics into our automated checks. I want to know certain things, like I mentioned, how long it took, how many clicks was in that check? If you've got a check that's got nearly 25, 50 clicks or something like that, is that a good check? Should that be broken down into one, two, three, four, five different checks? Does it have to be this monolithic thing? Why am I doing so many clicks?Again, adding this kind of information can help us look at the health of our automated checks.

Then we've got the logs and screenshots. For example, if a check does fail, just write a little script that goes and pulls the logs off the server that it's running against and download them, take a screenshot of the end result, or take a snapshot of the JSON. Save the JSON to a file or save the XML, and then just ZIP that up and stick it on there as a test artifact so that we can then use that afterwards to investigate what went wrong, and at the same time though they also serve the purpose of that evidence bit I mentioned. Think about adding some of this stuff into your code base. Again, I've used every one. I've used every one because of different reasons.

For example, with your testers and your automation engineers, we want to know what's going on with our checks. We want to know the health. You've then got your product owner who's using these results to help them make decisions, so they want to being able to find the information that they need, and they might want it in a different format to you, so you might have to consider that. What do your reports actually look like? Perhaps you're in quite a micromanaged environment where you've got to produce a weekly test report and it has to be in a certain format. Perhaps the management want rack statuses. Can you code that into your automated checks to produce that automatically for you, instead of you wasting the 30 minutes, an hour every week trying to piece this report together from looking at the results in Jenkins or whatever CI you're using? Got to think about this reporting and build that into our automated checks, build that into our architecture.

Then some questions to go and ask yourselves. Whenever you do have a failed check, how long has it taken you to go from knowing it's failed to fixing it to having it back in and having it being green again? Can you improve that? Do you need to monitor certain things to help you with that? I know a check's actually healthy, again, not just because they're green doesn't mean they're good. Again, your state could be perfect. Your algorithm could be perfect. Your oracles are spot on, but if it's taken a minute to run every time, and realistically if you do it yourself as a human and you could get through that process in 20 seconds, I would argue your check isn't that healthy. Let's keep refreshing, let's keep looking at that. Let's add some reporting to help us do that.

What do your checks actually cover? What coverage do you have? Can you add something into your reporting, perhaps into your test framework that you're using that will potentially give you some coverage? Now, coverage is a lengthy topic, and again I could probably rant about coverage for a while, but for me coverage is coverage of some model. Can you build something that basically models how you view the application, map all your automated checks to that model and then get an insight into the coverage that you have? Again, that's going to help you when the team come in and they're going to talk about a new feature. "We're going to rewrite this. We're going to refactor this." You can have a look at at your reporting, have a look at your coverage model and say, "Yeah, we've got quite a lot of coverage there. We've been doing continuous reviews of our oracles, so, yeah, I reckon ... "

There's risk obviously with refactoring but we can mitigate that with our automated checks, and we can do all that by thinking about reporting. It's often overlooked in my opinion. We tend to just rely on your assertion passing or failing. A lot of people tend to rely on the message as well that you can stick at the end of your assertion. "I was expecting one, but I got two." Again, that's a terrible assertion message. Adding some more information into that is a good start. For example, I was expecting one order but there was ten orders. That makes it more contextual, instead of just getting an error message that says, "Expected one but got two." Again, reporting, very important.

Two angles to it though. Information about the check. Information about what it actually means the fact that it failed or passed, or the fact that it didn't detect change or did detect change. That's the R. That's reporting.

We're up to E. E is for execution, but before we look into execution did anyone find anything wrong with my previous slide or my previous whiteboard [inaudible 00:40:00]? If you've noticed, thinking about oracles again, you would have seen that it said SCARED that time instead of SACRED. Some of you would have picked it up, and that's my point. Oracles are always changing. We're always finding new ones. Anyway, let's look at execution. By using the word execution what I'm talking about is where we execute these checks, which environment we're running them on and how are we going about doing that. Are we running them in parallel? Are we running one at a time? Are we running a block? Are we only using 10? Do we have some tagged as smoke test and we do them at a certain time? Are they running on a CI and perhaps that impacts the way they run? Are we running them in a headless Chrome or using XVFB? How are we thinking about execution and executing them?

We have to think about this because it impacts the way we design and it impacts the way the CIs are built and it might impact continuous delivery if you're in such a context. We need to think about it to maximize their value. We don't want to necessarily write automated checks that are tied to an environment. Perhaps they're tied to the QA environment or the test environment. You want to be able to write them so we can reuse them wherever we need to, and therefore we have to factor that into our design. It

might impact our algorithm. If we are tying them to a specific environment, perhaps that environment doesn't have as much hardware as another one, therefore our waits are longer.

Perhaps that environment's a super-speedy one, therefore our waits are lower and perhaps we might run into issues if we do it on another box. The same with timeouts on various API calls, for example. Is there any external impact? If we are running on the QA environment or the test environment, as I know a lot of people do, who else is using that environment? Could someone go onto that environment and delete your data? Could your checks put some data in place and then someone comes along and steals it or views it? I've had numerous checks fail because someone's [inaudible 00:42:08] and changed something. Perhaps they've changed the config on the server that you were executing against, and therefore you've lost the behavior, and that goes back to the state we spoke about earlier on, being able to control that state.

If someone else can come along and interfere with that, we need to be able to either mitigate that or have some kind of feature or message in place that says the automation is running, don't do it. Where we're getting to these days, as we're going to look at further on, we shouldn't be worrying about environment sharing. We should be able to just spin up new ones on the fly whenever we need them to. We need to do this before we do our check implementation. If you are potentially looking at running checks in parallel or making them used on different environments, you have to design them to be so. Going back in and adding this later on is possible but it's very time consuming, so therefore let's try and do that from the beginning.

In terms of execution these days let's first look at the test frameworks. Now, are you going to run them in parallel? If you want to run them in parallel, does your chosen test framework support that? Does your CI support that? Do you have enough capacity on your boxes to support this? You've got to think about that. You've got to factor it into your decision making. In terms of things like Docker and virtual machines now, are you designing it to be able to just spin up on any machine? Therefore, if it is spinning up on any machine, you might have to have some environment variables such as the URL that you might be hitting or the API address. You might have to put them into config files to avoid having them hard-coded. You've got to think about that as you're implementing your checks.

Are they going to be running on CI? Where are they going to run on that CI? What agent are they going to pick? Is that box beefy enough? Does it have all the prerequisites that you may need for your execution? Does it have the browsers installed, the right versions? Does it have all the libraries that you made it dependent on? This is where we now move into another realm when it comes to automation engineers and testers. CI in pipelines are backing a huge thing, and we need to be aware of it because those things are exercising stuff that we've been building and therefore we need to look at how they work, get help from the teams. Devops is a buzzword these days. Get some help from your DevOps engineer if you have them to make sure that you're designing

checks to be able to get the most out of them on these pipelines and on these CIs.

Not everyone's involved in this. Your product owners probably don't really care where they're being executed, but DevOps testers and devs can help you with this. Thinking about about the execution. It's okay having all these checks, but do you always have to run all of them? Perhaps we should tag some of them to only be run on smoke, to smoke checks. I want to run these 15 first after every single build or deploy, because I want to know if anything fails quickly. They could be your most critical paths, you know revenue. Anything that generates revenue is important. Anything that's about brand reputation. There's a really good mnemonic: RCRCRCRC from Karen Johnson. Perhaps you want to factor that into your execution.

Perhaps you want to work on areas that have been recently changed first. If you go back to the reporting and you have this kind of coverage model in place and you know that most of the commits are in a certain area, perhaps you want to have some module that says run all these first, then run everything else because you want that feedback quicker. You want to know immediately whether any of those are failing because that potentially could be where you're expecting it. Executing is really important when it comes to our automated checks.

At the same time, you want to be running them regularly as well. There's no point writing these things if you're going to run them once a week or overnight. Hardware is cheap these days. Just running them as much as you can, get as much value out of them. Firstly you're going to get more feedback on your actual product, but you're also going to get more feedback on your actual checks. You might start seeing that some are failing more than often. Again, if you go back to the R and you've got good reporting in place, you can improve that over time and you'll get more data the more you run them. They're cheap. Hardware is cheap. Run them as frequently and as much as you can.

Some questions to ask yourself. Could you run your checks on any environment? If someone turned up in your company and just spun up a new environment tomorrow, would your checks execute on that environment? If not, why? Do you know why? Are you aware? Do you have enough knowledge of your product and of your suite to be able to understand your architecture, to understand why those checks won't run on that environment? Are any of your checks tied to a specific environment? If they are, is that okay or does that need changing? Perhaps it was a decision made a long time ago and hasn't been revisited. Let's have a think about that. More importantly here, we want to run our checks in parallel because we want that feedback quicker. That's what having automated checks is about, continuous, fast feedback.

Now, to do that your checks need to be check safe, for want of a word, making a play on the word thread-safe. We don't want any checks to bleed into other checks. We want them to be self-contained and be able to execute all on their own. If they're not, then you might run into some issues where you've ended up

with dependencies on your checks. Check A, B, and C has to run before D, otherwise D will fail. Perhaps it doesn't have the right data in place. Perhaps it's using data that was set by A, B, and C. If that is the case, let's look at refactoring those and jump back to the S. Think about the state. Can I do that in a different way so that I don't have any dependencies, and therefore if suddenly I do have unlimited hardware, I can run all my hundred or thousands of checks in parallel and you can get feedback within a few minutes instead of spending hours or overnight as I know some teams still have.

That's execution. Very important. We've got to run these things a lot, so we need to make sure we've designing them to be able to run as frequently as we like, as stable as we like. One to go.

The final letter. The final letter is D, and in this case the D is for deterministic. Deterministic is basically the goal, it's what we're trying to achieve. We want our checks to be deterministic because we want to know exactly what they're going to do, how they're going to do it and what that means for when it passes or fails or detects change. We want to know exactly every single time that it's going to do the same thing. We want it to be deterministic. Reasons why. Reliability. We want these checks to be reliable. We're using them to get fast feedback. We're using them for information about the product, about the quality, insights into that product. Information that we then subconsciously sometimes use to dictate our further testing.

If everything's green, I can probably guess that some of you will not do as much exploratory testing as you may have done. If it goes red and it's been red on several occasions, you're probably going to go and do some more exploratory testing, or you're going to be a bit more nervous about it. In that situation we want to know exactly what they've done and they've done exactly what we told them to do, how we coded them, how we made our algorithm, make sure our oracles are solid, and that we have all that reporting in place.

We need to trust them. They're a huge part. Companies and teams are investing a lot of money in these automated checks, and you need to trust their results. If you don't trust their results or you're ever within any doubt, it's probably because they're not deterministic, and if they're not deterministic perhaps the SACRED model can help you to go through and do some analysis on the checks that you have, and see if you can spot any areas to improve. When? All the time. Pretty much all the time. Every time we add something to our architecture, to our checks, when we think about what we're going to build and what we're going to implement, we need to be thinking all the time in the back of our head, "Is this deterministic? Is it going to do the same thing every single time?" Because that's what we wanted it to be.

If there's an area or some part of the state that we haven't controlled, that may be a risk to meeting this target of always being deterministic. If we've added some if statements, a few dodgy waits into our algorithm, it's not ideal, pulling

back to our deterministic and to deterministic ability. We need to think about this.

Again, as I said, we do testing all the time. You need to test your checks. Sounds crazy, right? Who tests the tests? We can get into that turtle analogy, turtles all the way down, but it's true. Well, building a piece of software that's what often is overlooked and that surprises me about the industry. We've got product owners, developers and testers, analysts, DevOps teams for our products, but a suite of automated checks and automation architecture is a piece of software, and usually done by one or two, three, four people that all have the same role and that role is to be an automation engineer or a tester. Sometimes we've got to put on a different hat and look at ...

I'm saying this is my model. This is what I've been using, but this allows me to think about different aspects and put on that different hat to think from a slightly different angle, because we're building a piece of software and we normally have teams of 10, 15 people working on the actual thing we sell. Sure, perhaps it's not as complicated, it's not as complex, but it's still a piece of software and we need to test it to make sure that we're hitting that goal of being deterministic.

Continuously review. We need to be going over everything, as I've said. Is the reporting right? Are our oracles good enough? Is our algorithm solid enough? We need to be continuously reviewing this to focus on making sure that when they go green or when they go red we know that there's a potential problem there and it's not just some flakiness. Everyone gets involved. Finally, some questions to think about this. When your build radiator is green, do you actually know what that means? Do you have enough confidence, enough trust in your checks to be able to go to someone and explain to them exactly what it means that it's gone green? If you don't, have a review of your checks, improve the reporting, try and find out why, and try and build that trust level up.

If I was to come to some of your checks and make a few changes, if I was to change A, would you be able to tell me what will happen to that check? If I change some data in the database, can you tell me what the results or what the outcome of that check will be? Will it fail and will it tell you exactly why? That's really important. Have a play around and see what you can find out. This isn't the question, but it's a good measurement. In your team, how many times are you saying the word flaky? How many times are you saying something along the lines of, "Yeah. It's done that before. It always does that. It's not important. It'll pass the next time we run it." Act upon those things. They are triggers. They are desperately telling you to come look at me, explore me, investigate me, try and find out what's wrong with me.

As far as I'm concerned, whenever a check goes red, it's an invitation to explore. Let's go and find out why it's gone red and see if I can identify the cause, improve it, change it, fix it, or whatever word you want to use so that I can make it deterministic, because that's what I need to achieve. I want reliable

automation that's fast to execute, gives me fast feedback, and when it does fail, gives me all the information that I need to know in order to be able to understand what the problem may be, if there is a bug, or if there is some important information I need to give someone, and I want to trust my checks to deliver that all the time. Deterministic is the ultimate goal, because otherwise we have flakiness, we have mixed results and you can end up in this really awful place.

I joked about this at the Selenium conference recently. You can have someone's job is they come in and fix broken checks. Literally come in Monday morning, build's red, make the build green. Go for lunch, come back, build's red. Make the build green. Go home. Come back in on Tuesday and so on, and so on, and so on. How depressing. How boring. You don't want to be doing that. We want to be testing. We want to be getting knowledge. We want to be building checks of value, so let's do that. That pretty much brings me to the end.

The SACRED model. As I said, I spent some time thinking about what I was doing as an automation engineer. Whenever I was writing some checks, what was I actually doing? I was doing significantly more than writing code. I wanted to understand what I was doing so I could share it with others, and since doing so I've been able to talk to many people about do you have to go down the coding route or can you still add value to automation? I'm hoping from watching this and thinking about some of the ideas that I've shared in here that you can get an understanding and appreciation for some of the other roles that exist in your team, how they can contribute to your automation efforts without having necessarily to know how to code.

Perhaps there's a few testers in your team that management might be pushing into the coding, trying to line them up to go to code school. Can they still be of immense value to your automation without having to go that code level? Can they pair with you? Can you mod on something? Don't have to go to the code school. Other people in your team, they can contribute to your automation efforts. Don't struggle on your own. Go and talk to the developers, the DevOps teams, your product owner. Try and get some help, some insight. If your introduction into automation was always down the code route, have a reflection on some of the other skills that I've mentioned. Can you improve them? Do you even know where to go to improve some of them?

Perhaps you need to go and investigate that. Perhaps using the SACRED model to go back and review, use it as a pattern. Perhaps you could give yourself a score. I thought about this the other day. Could you give yourself a score out of 10 for each of these components, and therefore give yourself some direction to go and improve on your checks that are out there? To finish, it's just a model. It's just my pondering, my thinking, but it has helped me so far and I know it has helped other people out there. A colleague of mine, Mark [Winteregan's 00:57:41] been able to use it in his company to redesign some existing checks. He spoke about that at the Selenium conference, as an example.

It applies to all levels of automation. This is not tied to UI. The concepts, it works on every layer that you would want to add automation in. Have a think. See if you can use it. I hope it brings you some value. Right now getting ready for some questions. I hope you can join me for the live Q&A. Please fire any questions to me. I love to answer them. To be honest, I think it's one of the best times because you can actually find out if a speaker does know anything or not. Thank you for watching. I hope you get some value out of it. As I mentioned, my name is Richard Bradshaw. I'm also the Friendly Tester. I blog as thefriendlytester.co.uk, and I'm also the boss at Ministry of Testing.

If you've not heard of Ministry of Testing, check us out. We've got a cool logo, and we're very supportive of the wider community. Thanks for Joe for the opportunity, and let's dive into these questions. See you soon.