SUBJECT7

QUALITY IS A CULTURE

Pivoting to Codeless Test Automation

By Matt Heusser for Subject7

At my company, Excelon Development, before we make a recommendation, we usually do an analysis with the Six Boxes Model. The boxes are performance factors, like incentives, management expectations, tools and process. The analysis includes determining if the people doing the work can do the job (skills) or are capable of learning to do the job, which we call *capacity*.

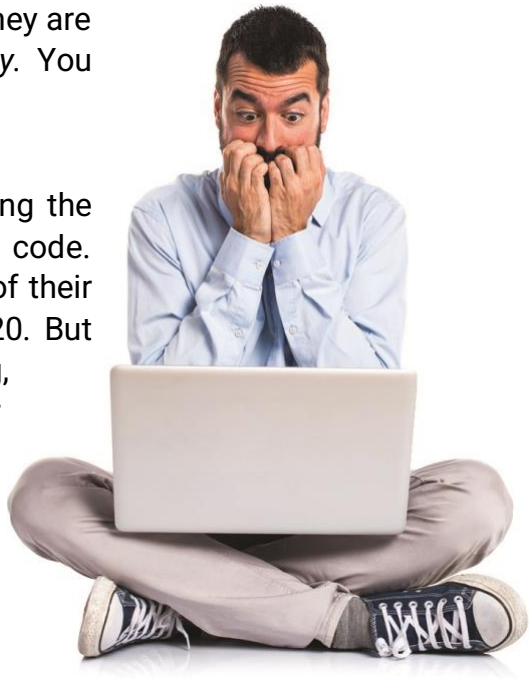
It's a tough lesson for some of us when we realize sometime towards the end of high school, that we do not have the vision required to be a fighter pilot, or the height to play professional basketball; likewise, the assumption that everyone can learn to code. I suppose most people that work in an office can learn to code … badly. A thousand abandoned test automation efforts tell me that we should at least consider whether our staff has the capability, and more importantly, the interest in writing software.

Every responsible article about testing tooling talks about getting the right people on the bus, on building the right team. Some even go so far as to suggest that you consider if the people you have (and the ones you have budget for) will match the test approach you are considering.

Today, I will go a little further than that, and talk about the skills that matter for a codeless approach to testing.

WHERE YOU START

Imagine a group of testers and analysts. They are *happy*. They understand how they contribute to the process, documenting the system, finding problems in the software before it is released, writing up bugs, perhaps acting as second-line support. For their education and experience they are compensated and respected well enough. Again: They are *happy*. You might have people very much like this in your office.



Then a typical automation tool comes in. Now, in addition to doing the testing and the analyzing, these people are expected to learn to code. Actual programmers, of course, learn to code in about one-fourth of their college courses -- about 30 Computer Sciences credits out of 120. But that's not this team, most of them took one course in programming, a decade ago. The company now asks them to learn to code in their spare time. The reason they need to learn to code, of course, is because testing is falling behind.

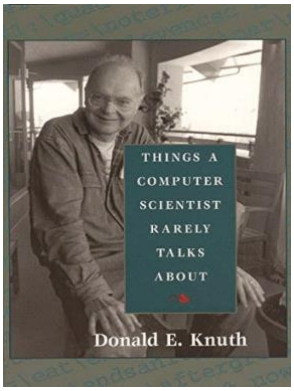
This is not going to go well.

It can be done better or worse. Out of a larger group, perhaps Thirty people, there might be three to five with the aptitude and motivation to learn to code on their own. If their job is at risk, perhaps ten or fifteen will be willing to try. At the end of the process, instead of respected experts who understand their niche, you're left with people who are coding a monstrosity beyond their skill, who are overworked, and now, thanks to the magic of programming salaries -- feel underpaid.

Let's flip the script and talk about what it takes to be successful with adaptive, low or, codeless solutions.

A DIFFERENT WORLD

Donald Knuth is known for pioneering, if not inventing, what we call computer science today. He talks about the skills programmers need in his book Things a Computer Scientist Rarely Talks About, including the ability to jump up and down many layers of abstraction. In code, that might be from a high level architecture, to a class, a method, and even the substance of the packets going over the internet - and back. According to Knuth, the mathematician looks for a single unifying truth, while the computer scientist can simply code differences with "if" statements.



Low and codeless frameworks don't typically have those layers of abstraction that programmers work with. Instead of doing many things, perhaps randomly, jumping around the screen, low and codeless frameworks do the same thing, over and over again. Click, Click, Click, Type, Click, Inspect, Compare -- that might be a simple tool-assisted test. Instead, a test automator using a simple tool-assisted-test needs to be able to create the test, look at it, understand that it matches the expected user behavior. They need to look at it and say, "Yes, this is what the user will do, and what they expect to see."

These kinds of tools might have an "if" statement, or a “for” loop. They might have a small amount of code, perhaps to calculate when a product will arrive, knowing it must be within five business days. This kind of tool jockey might need to go into a web browser and grab an image, or look in developer tools, or figure out the string that will locate a button so the tool can click it. None of this involves the kind of three dimensional chess skills that programmers need to have, keeping multiple variables in their mind at multiple levels.

There is a certain kind of magic to working in this way, because it connects two things: requirements and automated testing. Before the feature is coded, there are requirements that express what the software should do, then the automated tests demonstrate the software can do it, at least under some conditions. You can think of requirements and tests as bookends to the project, with the code in the middle. Writing test code can certainly work; I spent three years of my life at Socialtext working in Selenium every single day. What we didn't have at Socialtext is that bookend process. When I worked with analysts who stuck around to help with the acceptance testing thy didn't need to have an argument when the software didn’t do what someone expected; they knew. They *wrote* the requirements. They were ones who did the expecting.

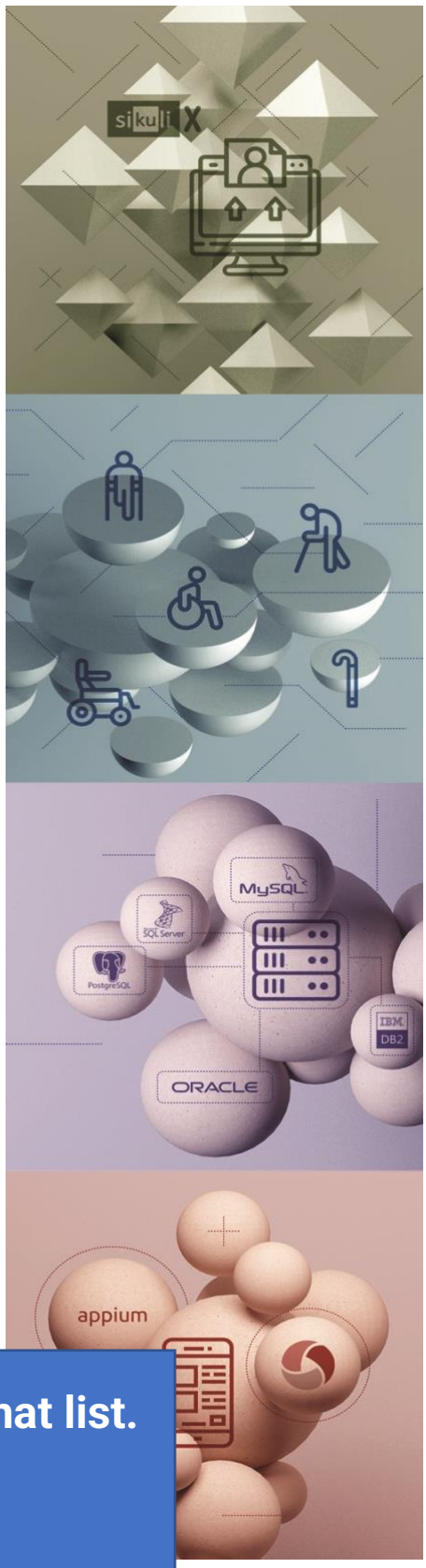
THE UNDISCOVERED COUNTRY

When I talk to people about software testing, they often fail to consider test design as an important skill. It is amazing. The assumption is that anyone can do testing. Then, when bugs slip through, we ask "Why didn't QA find that bug?" especially when the company offers no training on test design and assumes anyone can do it!

Experienced testers, the kind that find the important bugs, have learned test design. They likely understand what the user will do (mental modeling) and understand the history of what tends to break (regressions) and the kind of features that tend to “show up broken” for testing in the first place. These are important skills, but they only show up when the tester understands that one part of the role is exploring the software, and the other part is creating tooling.

In my experience, when testers are pushed to do “100% automation”, they often stop exploring and spend all their time automating.

Codeless tools can enable the bookend effect I discussed before. The "manual testers" can be more like subject matter experts who understand what the customer is trying to do, yet still produce working automation that runs. Designed well, with an eye toward test design, the automation can create an ever-increasing amount of coverage without significant maintenance effort. In other words, with codeless tools, it can be possible to have your pie and eat it too -- if the tool will truly work for you.



Before you jump into a test automation tool, take a hard look at the staff you have. Consider if the tool will marginalize their skills, force them to learn new skills, or augment the skills the testers already have. Is it going to create cyborg testers, who have cybernetic vision, hearing, and strength -- or is it going to put a mountain in front of them to climb. There are, of course, plenty of companies that have created a "new vision" of the future, with generous severance packages for testers that didn't want to learn to code. Some of them report great success. All of them report watching decades (or centuries) of experience walk out of the building, never to return.

If you have a group of testing experts that you would like to keep, and you would rather not create a technical test-automation-team, then you may be looking for a low-code solution where the automation "pops out" of the process at the end. That is, the testers have to test, and in the course of their work they need to run through the very same scenarios that could be automated. Why not use a codeless tool that can use the work that they already do to create automation, but with essentially codeless required?

The main objection is that codeless tools don't always work. You have version control issues and multi-user issues and object recognition issues and a host of other problems. Suddenly, the list of workable codeless solutions may not be as long.

Still, I would submit, Subject7 would be on that list.

Take a look.

SUBJECT7
QUALITY IS A CULTURE