The "A" Word

Under the Covers of Test Automation

Alan Page

The ”A” Word

Under the Covers of Test Automation

Alan Page

This book is for sale at http://leanpub.com/TheAWord

This version was published on 2013-07-30

This is a Leanpub book. Leanpub empowers authors and publishers with the Lean Publishing process. Lean Publishing is the act of publishing an in-progress ebook using lightweight tools and many iterations to get reader feedback, pivot until you have the right book and build traction once you do.

©2012 - 2013 Alan Page

Tweet This Book!

Please help Alan Page by spreading the word about this book on Twitter!

The suggested tweet for this book is:

I just got The ”A” Word by @alanpage #theAWord

The suggested hashtag for this book is #theAWord.

Find out what other people are saying about the book by clicking on this link to search for this hashtag on Twitter:

https://twitter.com/search/#theAWord

Contents

Introduction 1
Testing: Failing to Succeed 1
The Robots are Taking Over 3
To Automate …? 5
The Coding Tester 8
GUI Shmooey 10
Design for GUI Automation 12
It’s (probably) a Design Problem 17
In the Middle 20
Test Design for Automation 22
Orchestrating Test Automation 27
LOL — UR AUTOMASHUN SUCKZ! 29
Exploring Test Automation 31
Musings on Test Design 35
Beyond Regression Tests 39
Testing with Code 42
More on Test Design 45
Coding, Testing, and the A word 47
Last word on the A word 50
Resources 51

Introduction

Thanks for reading, and perhaps even purchasing[1] this little e-book on Test Automation. To be clear, this isn't a typical test automation book. There's little to no talk of tools or frameworks - just a lot of ideas, a bit of philosophy, and a few opinions thrown here and there. This is definitely **not** a how-to book (but it may indeed be a ***how-to-think-about-testing-and-test-automation*** book).

In fact, I lead this e-book off with a post I wrote about The Five Orders of Ignorance — which has nothing directly to do with automation at all. The orders of ignorance are a wonderful model to describe *knowledge acquisition* - something at the core of nearly every testing effort.

Almost everything here was originally written on my blog[2], but I've edited extensively, added clarification in a few places, and tried to reorder things in a way that makes (a little bit of) sense. I edited to remove some obvious repetition, but some themes (opinions on automating the GUI, coding approaches for testers, etc.) remain. I expect that the points I feel the strongest about are the points that are repeated the most.

Many (most?) of my blog posts on automation center around the abuse and misuse of automation in testing and software development. As I gathered some of my old blog posts on the topic, I realized that just about everything I've written about test automation shares this central theme. I, for one, have never bought into the "automation as a time saver", or "automation to replace manual testing" thoughts I hear from time to time. Sometimes, testers use programming skills to help their testing. Sometimes, that code automates some application functionality. That's it. The notion

[1]All proceeds from the book (90% of whatever you paid minus fifty cents) will be donated to The American Cancer Society. My mom died from cancer in October, 2012, and besides bringing me into this world, there's little that I've accomplished in life that I don't credit to her.

[2]http://angryweasel.com/blog

that the activity of testing is comprised of walking through a specific set of known user tasks is ludicrous.

But more on that (much more) are in the following pages.

At this point, I should add that I also included a few notes on test design. These aren't filler - test design includes determining what testing you're going to do with code - in *addition* to human testing approaches (points often forgotten or ignored when many people practice testing). In my opinion, test design (and product design, for that matter) *includes* determining all of the what and how of testing - including what will be automated, and how it will be done.

I have dead-tree published works on automation as well. I wrote about some of the automation approaches at Microsoft in *How We Test Software at Microsoft*; Large-scale test automation in *Beautiful Testing*; and on Device-simulation frameworks in *Experiences of Test Automation: Case Studies of Software Test Automation.*

I hope you enjoy reading my ramblings. If you want to contact me, you are always welcome to email alan@angryweasel.com, or through my web site.

Testing: Failing to Succeed

> ”…the paradox of testing: We want to find defects in the software under test, but we do not really want to find them”. - Phillip Armour

I often speak of Phillip Amour’s Five Orders of Ignorance[3] as they relate to software engineering as I think they’re a wonderful model of how we acquire knowlege (including how we acquire testing knowledge, as well as knowledge about the product under test).

First, for those unfamiliar, let me briefly cover the orders of ignorance.

0OI – Zero-Oh-I is *lack of ignorance.* It’s when you know something. I know, for example, that the first track on the Rolling Stone’s *Sticky Fingers* album is *Brown Sugar.*

1OI – *lack of knowledge.* I don’t know (or remember) what the track is following *Brown Sugar*, but I could find the answer quickly.

2OI – *lack of awareness.* You have 2OI when you don’t know what you don’t know. I know that there are Stones tunes that I’ve never heard before, but it would be impossible for me to make a list of Stones songs I’ve never heard before.

30I – *lack of process.* You have 3OI when you don’t have a suitable method for discovering 2OI (for discovering what you don’t know you don’t know).

4OI – _meta-ignorance.** **_You have 4OI when you don’t know about the five levels of ignorance.

In an article I wrote for Better Software, I attempted to stress how important the levels of ignorance are to software testers and software testing, and I frequently bring them up in conversations about testing. Recently, my mind was blown when Phillip Armour

[3]http://www-plan.cs.colorado.edu/diwan/3308-07/p17-armour.pdf

wrote an article specifically about the application of the levels of ignorance to software testing. A direct link (for those who have access to the communications of the ACM) is below[4].

When we test to ensure requirements were implemented, or the stories function as expected, we are testing for 0OI. We know what pass and fail looks like, and we can create tests accordingly. The 0OI testing set may be large, but it's bounded. A passing 0OI test doesn't expose any new knowledge – it just proves what we thought we already knew.

Testers also perform a lot of 2OI testing. We explore or run dynamic tests on a system to help discover what we don't know we don't know. Testing for 2OI is an unbounded test set – the things that a system *might* do is infinite. This is where we need to devise tests *when we don't know what we're looking for*. (Note – we don't test for 1OI because if we truly knew in advance what we didn't' know, we'd resolve the ignorance and conduct 0OI testing.

Another (shorter) way to think about this is that 0OI tests are *knowledge-proving* tests, while 2OI tests are *knowledge-acquiring* tests.

Armour goes on to discuss the theoretical information content for a test and optimal failure rates (which I'll leave to you to read, as I've probably already butchered the content already). It's good stuff full of great quotes, and I hope you check it out.

---

[4] http://dl.acm.org/citation.cfm?id=2001280&CFID=49086211&CFTOKEN=83382740

The Robots are Taking Over

The local company that sells everything under the sun just bought a robotics company (Amazon bought Kiva systems).

Normally, I don't write about news stories, but on the way to work this morning I heard an interesting discussion on a local talk radio station. It turns out that there are people up in arms over the purchase because using robots in the warehouse puts people out of jobs. To be clear, I don't like people losing their jobs either – but bear with me for another paragraph.

The arguments continued with comments like, "The robots can't do all of the warehouse work – sometimes you need to use your brain to find a misplaced item", or, "You will still need people to verify *with their eyes* that the right items were selected". Others countered with comments like, "The robots can work 24 hours a day", and, "You'll still need people to program the robots and give them directions".

I laughed out loud in my car as I realized that these people calling into the radio station were having the same discussion many people have about test automation.I won't rehash or expand, but I will summarize with two blurbs of less than 140 characters each:

> **Humans fail when they don't use automation to solve problems impossible or impractical for manual efforts.**
>
> **Automation fails when it tries to do or verify something that's more suited for a human evaluation.**

Since I didn't use the word 'test' in those tweets, I think my comments apply to Amazon's (probable) warehouse changes as well.

I was happy to hear that a few callers recognized that robots (and automation) can be used to solve left-brained monotonous work – and do it pretty well. The future of work is (IMO) creative work[5], and (again, IMO), automation (and robots) are what we need in order to enable us to find the time and insights to develop our creative thoughts into game-changing innovation.

And that sure beats doing the boring stuff.

[5]Daniel Pink wrote a whole book on this concept: *A Whole New Mind: Why Right-Brainers Will Rule the Future*

To Automate ...?

To Automate, or not to Automate, that is the question that confuses testers around the world. One tester may say, “I automate everything, I am a test automation ninja!”. Another tester may say, “I do not automate, if I automate I fail to engage my brain properly in the testing effort!”. Another may say, “I automate that which is given to me to automate.” Yet another may say, “I try to automate everything, but what I cannot automate I test manually.”

All of these testers are wrong (in my opinion, of course). So wrong, that if I had the power, I’d put them in the tester penalty box (hockey playoffs on my brain) until they came to their senses.

Good testers test first – or at the very least they think of tests first. I think great testers (or at least the testers I consider great) first think about how they’re going to approach a testing problem, then figure out what’s suitable for automation, and what’s not suitable. I say that like it’s easy – it’s not. One of my stock phrases regarding the approach to test automation is, “**You should automate 100% of the tests that should be automated**”. Nearly every good tester knows that, but finding the automation line can be tricky business.

I have my own heuristic for figuring this out – I call it the “I’m Bored” heuristic. I don’t like to be bored, so when I get bored, I automate what I’m doing. When I’m designing tests, I try to be more proactive and *predict* where I’ll get bored and automate those tasks.

Just today, I was fixing some errors reported by a static analysis tool and found myself doing the following.

- copy the file path from the log
- check out the file from source control
- load the file in an editor
- make the fix (or fixes)
- rebuild to ensure the error was fixed

After the third time, I was bored. I spent the next two minutes and fifty-one seconds writing a doskey macro that would pull the file name from the log, print it to the console (for easy copying), check out the file, and finally, load the file into the currently open editor. Given that I had another 40 files to go through, I consider that a good automation investment.

In *How We Test Software at Microsoft*, I told the story about my first week at Microsoft. My manager handed me a list of test cases (in Excel) and told me to run the tests. I glanced at the list of eighty or so tests and asked when he expected me to finish automating them. He said, "Oh no, we don't have *time* to automate those tests, you just need to run them every day."

As an aside, let me say that I think scripted manual tests are one of the most wasteful efforts any tester – no, anyone in the world – can take on. I know that some readers will cite examples of where manual scripted tests are valuable, but for the record, I despise them more than my daughter despises brussel sprouts (you don't know my daughter, but you may have heard her screams of dismay across the country the last time I tried sneaking a few on to her plate).

So anyway, I started work on a Monday, got bored running those tests by Tuesday, and automated all eighty tests on Wednesday. I used my new found spare time to find all sorts of other issues (and to automate other scenarios). I don't know if I ever told my manager that I automated those tests, but he was pretty dang happy with my results.

Not all automation efforts work this way (I'm talking to you, pointy haired manager). The dream panacea of automation for some folks is that testers will write a plethora of automation, then they'll have time to do all kinds of additional testing while the automation runs seamlessly in the background. If all automation efforts were created equal (and by that, I mean equally simple), and if testers took the time to write reliable and maintainable code, this *could* be possible, but I don't know anyone that lives in that world. Some things are

difficult to automate, and we can waste our time trying to automate those tasks. Sometimes we write fragile tests because they appear to work (the illusion of progress). Then reality sets in and we discover we're spending a big chunk of our time investigating and fixing failed tests (but that's another story).

My parting advice is to remind you all that as software testers, our job is to test software. That probably sounds stupid (it sounds stupid to me as I type it), but test automation is just one tool from our tester toolbox that we can use to solve a specific set of testing problems. Like most tools, it works extremely well in some situations, and horribly in others. Use your brain and make the right choices.

The Coding Tester

Every day, it seems, I come across an article, a forum posting, a blog, or a tweet bemoaning the end-of-test-as-we-know-it because some company has hired / is hiring people with programming knowledge to become testers. I've written about coding-testers before, and I think most folks recognize that *in some (and no, not all) situations*, having testers who code can be an advantage.

It's important to note that the role of a coder-tester is NOT to automate everything (instead, of course, you should automate 100% of the tests that *should* be automated). Writing automation is **one** task of a coding tester, but certainly not the **primary** focus (yes, I know that in some companies, automation "experts" only write automated tests. In my opinion, these people are not really testers, so they don't count).

I, my colleagues, and many of the best testers I know are good testers first (well, at least my colleagues are), but have programming chops that they can use to solve difficult testing problems. Automating a bunch of rote tasks (assuming robustness, accuracy of the verification, maintainability, and numerous other attributes) *may* be a difficult testing problem, but it's just *one* testing problem. Good testers simply use the tools in their toolbox to solve the problems and challenges they run into.

My head isn't in the sand though – I know that there are a lot of people hiring testers out there who are thinking, "I want to hire a bunch of coders to do my testing, because then they can automate everything!)." That, of course, is stupid, and I'm sorry that situation exists. If you run into these folks, feel free to send them my way, and I'll be happy to explain a thousand other ways programming knowledge can help someone be a better tester, and why attempting to blindly automate everything is pure idiocy.

To be clear, I do **not** think that all testers need to have a computer science degree (for the record, I have a graduate degree in music

composition). I also don't think testers **must** be able to program. It depends completely on what you want testers in your organization to do. It's certainly possible to do great testing without using any programming skills – you just need to think about the role you want testers to perform (or what you want to call testing), and hire the people who fit that role.

On the other hand, I find it a bit silly to think that just because someone knows how to program, that they are somehow "tainted", and will be unable to look at a program from the proper "tester angle" because they have a notion of how the stuff under the hood actually works. I've worked with a hundreds of testers who can code, and hundreds more who could not. In fifteen years of working with testers from both camps, I've seen no correlation of increased customer empathy or testing talent emerge from either camp. My study is only anecdotal, so if there are specific studies in this area, please point me to them.

A final thing to remember for anyone who thinks they're somehow closer to the customer due their background (or lack of a background) is that you are not the customer. It doesn't matter if you're a former bank CEO testing a financial suite – you're still not the customer. I think it's critical to use whatever means you have to *learn* about the customer, or to *understand* the customer, but you will never *be* the customer. Regardless of the skills you do (or don't) have.

GUI Shmooey

I answered a few questions recently about automating GUI tests. One question was about recommendations for GUI automation tools for non-coders, and the other was about how much time to spend on the GUI in an MVC (model-view-controller) application. The answers were easy. In the first case, I said that they weren't going to get ROI from the effort, and they should just test the GUI manually. In the second case, I suggested that they do all of the automation *ignoring* the view/GUI, and test the GUI (view) manually. I could expand an entire post on why I gave those answers, but it doesn't matter. I'm going to go out on a limb and make the following statement. **For 95% of all software applications, directly automating the GUI is a waste of time**. For the record, I typed 99% above first, then chickened out. I may change my mind again. The point is that I think testers, in general, spend too much time trying to automate GUIs. I'm not against automation – just write automation starting at a level below(*) the GUI and move down from there. I'm not saying that you shouldn't test the GUI at all – I just don't see why you wouldn't want to test it manually, and get people knowledgeable in user experience to help. I just think that in most cases we are wasting our time when we try to automate GUIs, and wonder if anyone has the guts to stop.

Just the GUI?

An unnamed commenter asked me once, “What if we ***only*** have a GUI?”. While I understand the context of their question, in my opinion, the only time you *only* have a GUI is if you are testing a picture. The code that turns the GUI into a useful application is testable - perhaps only testable through unit tests, but it is testable. But a bigger point is that manipulation the UI (or user actions), isn’t the only way to leverage code (or test automation, if you must call it that). If your app connects with a service, you can test those service calls extensively with a small program (or batch file). If your “GUI only” application saves or reads from files or a database, you can test that part *extensively* with a bit of code or scripting. There’s always something beyond the GUI. Perhaps you just need to look harder?

Design for GUI Automation

Now that I've stated my distaste for GUI automation I should share some clarification.

First off, let me state my main points for disliking GUI automation:

- It's (typically) fragile – tests tend to break / stop working / work unsuccessfully often
- It rarely lasts through multiple versions of a project (another aspect of fragility)
- It's freakin' *hard* to automate UI (and keep track of state, verify, etc.)
- Available tools are weak to moderate (this is arguable, depending on what you want to do with the tools - I'm particularly pleased, for example, with what good testers are able to do with selenium and web driver).

For now, let's ignore the last point. Partially because quality of automation tools is a debatable point, but more importantly because it's very much a potentially *solvable* point (one potential improvement would be if vendors would market these tools as test authoring tools rather than record and playback tools).

The first two points above have more to do with test design than the authoring of the tests. It's possible to write robust (or at least less fragile) tests that work and provide value as the product evolves. Designing tests with abstractions and layers and other good design principles will help immensely. Doing this correctly drives the third point home. Designing robust GUI automation is difficult. Of course it can be done well, but I'm not yet completely convinced it's worth the ROI.

Let me also acknowledge the fact that many people manipulate the GUI through code by manipulating the underlying controls rather

than the GUI *directly* (i.e. `Button.Press()` or `SendKeys("username")` type commands. I am vehemently against the latter, but tolerant (to an extent) of the former (depending on implementation of the underlying model / controller, as well as the design and intent of the tests).

That's why it's important when designing *any* test to think about the entire test space and use automation where it helps you discover information about the application under test that would be impractical or expensive to test otherwise. I hate loathe automated tests that walk through a standard user scenario (click the button, check the result, click the button, check the next result, etc.) But I love GUI automation that can automatically explore variations of a GUI based task flow. The problem is that the latter solution requires design skills that many testers don't consider, don't have, or don't think about.

Another example where I like GUI automation is in stress or performance issues. A manual test that presses a button a thousand times is out of the question, but it's perfect for an automated scenario. In fact, I wrote about something similar in *How We Test Software at Microsoft*.

> **Brute force UI automation**
> In most cases, UI automation that accesses controls through a model or similar methods tests just as well as automation that accesses the UI through button clicks and key presses. Occasionally, however, automating purely through the model can miss critical bugs.
>
> Several years ago, a spin-off of the Windows CE team was working on a project called the Windows Powered Smart Display. This device was a flat screen monitor that also functioned as a thin client for terminal services. When the monitor was un-docked from the

workstation, it would connect back to the workstation using a terminal server client.

The software on the device was primarily composed of core Windows CE components, but also contained a simple user interface that showed a history of connected servers, battery life, connection speed, and any local applications, as shown in the following graphic. The CE components were all well tested, and the small test team assigned to the device tested several user scenarios in addition to some manual functionality testing. Toward the end of the product cycle, I spent some time working with the team, helping them develop some additional tests.

One of the first things I wanted to do was create a few simple tests that I could run overnight to find any issues with the software that might not show up for days or weeks for a typical user. There was no object model for the application, and no other testability features, but given the simplicity of the application and the amount of time I had, I dove in to what I call brute force UI automation. I quickly wrote some code that could find each of the individual windows on the single screen that made up the application. I remember that I was going to look up the specific Windows message that this program used, but I'd hit a roadblock waiting for access to the source code. I've never been a fan of waiting around, so I decided to write brute-force code that would center the mouse over the position of the control on the screen and send a mouse click to that point on the screen. After a few minutes of debugging and testing, I had a simple application that would randomly connect to any available server, verify that

the connection was successfully established, and then terminate the terminal server session.

I configured the application to loop infinitely, started it, and then let it run while I took care of a final few odds and ends before heading home for the day. I glanced over my shoulder once in a while, happy to see the application connecting and disconnecting every few seconds. However, just as I stood up to leave, I turned around to take one final look at my test application and I saw that it had crashed. I happened to be running under the debugger and noticed that the crash was caused by a memory leak, and the application was out of a particular Windows resource. At first, I thought it was a problem in my application, so I spent some time scanning the source code looking for any place where items had been using that type of resource or perhaps where I was misusing a Windows API.

I couldn't find anything wrong but still thought the problem must be mine, and that I might have caused the problem during one of my earlier debugging sessions. I rebooted the device, set up the tests to run again, and walked out the door.When I got to work the next morning, the first thing I noticed was that the application had crashed again in the same place. By this time, I had access to the source code for the application, and after spending about an hour debugging the problem. The problem turned out to not be in the connection code, but in the graphics code. Every time one of the computer names was selected, the application initiated code that did custom drawing. It was just a small blue flash so that the user would know that the application had recognized the mouse

> click, much like the way that a button in a Windows-based application appears to sink when pressed. The problem in this application was that every time the test and drawing code ran, there was a resource leak. After a few hundred connections, the resource leak was big enough that the application would crash when the custom drawing code executed.
>
> I don't think I ever would have found this bug if I had been writing UI automation that executed functionality without going through the UI directly. I think that in most cases, the best solution for robust UI automation is a method that accesses controls without interacting with the UI, but now, I always keep my eye on any code in the user interface that does more than the functionality that the UI represents.

To be clear, I am not completely against GUI automation, but I do think that it's easy (too easy) to get wrong, and that it's overrated and undervalued among may testers. The best part about my opinion is that if you disagree with me, you have to write *good* GUI automation. In that case, we all win.

It's (probably) a Design Problem

As you know by now, I occasionally rant about GUI automation; but I don't think I've done a good job explaining some of the reasons why it *can* be fragile, and when it can actually be a good idea.

Let's take a deeper look at some of the attributes of the GUI automation design challenges.

- **Record & Playback automation.** This is a non-starter for me. I've never seen recorded automation work well in the long term– if your automation is based on record & playback, I can't imagine it being successful. Yes, the tools are getting better, but the real problem is …
- **What are you going to do about the oracle?** – Regardless of whether you write the automation, or you record it, simple playback of a user workflow has an oracle problem. The failure model in many UI tests is that nothing "bad" happens – the problem is that you don't often know what flavors of "bad" you're looking for, so your tests may miss stuff. **However**, *if* you *design* your tests with a trusted oracle (meaning if the test fails, you *know* there's a problem, and if it passes, you *know* the scenario works), you probably have usable UI automation. By the way – if your oracle solution involves screen comparisons, I think you're heading down a scary path – please consider this solution as a last, end-of-the-world type solution.
- **Bad Test Design I** – Many authors of UI tests seemingly fail to realize they're writing automation. Basic verification that would be hit by anyone walking through the basics of the application isn't worth much to me. However, if your automation actually *takes advantage of automation* – meaning that loops, randomness, input variations, and loads of other ideas are part of the automation approach, the approach may be worthwhile.

- **Bad Test Design II** – I dislike tests that do exactly the same thing every time. While valuable for regression testing, “hard coded” tests have severe limitations. In notepad, I can open a file by clicking the file menu, then selecting open, or I can press Alt-f, then o on the keyboard, I can press Ctrl-o, I can drag a file onto an open instance of notepad, or I can double click a file associated with notepad. A *well-designed* notepad test could just state File.Open(*filename*), and then use a randomly selected file open method. Too much UI automation doesn’t have this sort of abstraction and limits the effectiveness of the approach.
- **Bad Test Design III** – Lack of forward thinking is another design flaw I see often. I know of many GUI test suites that ran (successfully) on over 20 different localized versions of Windows 95, including RTL languages – *and* it ran successfully on Windows 98. Unfortunately, not all test authors consider the variety of scenarios where their tests *could* run.
- **Bad Test Design IV** – Failure to consider what can fail. This one irks me, because testers *should know that bad things can happen.* Consider what will happen when someone changes window text, a control type, or dialog layout. Consider what happens when Windows reboots your machine in the middle of a test run. You don’t necessarily have to make your test resilient to these changes, but at the very least, you need to make the error text point to exactly what changed and caused the error. Too often, tests fail and give zero information on why they failed. Plan for failure and ensure that all test failures tell you *exactly* what is wrong.
- **Bad Test Design V** – As a UI tester, you should have some notion of what sorts of UI automation are reliable, and which sorts are flaky. Using SendKeys works…but it probably should be a last resort. Good UI automation means that you know at least three ways to accomplish any task, and know which of the approaches is most reliable and which is least

reliable.

- **Bad Test Design VI** – One of my test design smells is Sleep (or similar) statements. The more I see in your test code, the less I trust it. Repeat after me – "Sleep functions are not a form of synchronization". Most frameworks have an alternative to Sleep. There is always a better alternative to Sleep statements.
- **Fragile UI** – It's really easy to write automation that fails anytime the UI changes. It's also really easy to write UI that breaks automation. If you don't start testing until late in the product cycle, and know that the UI automation won't be used in subsequent versions of the product, an investment in UI automation may make sense (given that it solves a real testing problem. Alternatively, if you're involved early in the product cycle, you could *wait* to write UI automation until the UI was complete. A third (and recommended) approach is to make sure that the application under test is *designed* in a way that makes UI automation more robust (i.e. increase testability). Testability is, in short, the expense of test. Rewriting automation frequently during the product cycle is expensive. Writing complex oracles to verify things that *should* be easier is expensive. Working with developers to implement a UI model that enables straightforward and sustainable automation is cheap – especially when you consider the long term benefits and gains. I probably have another post on this subject (testability) alone.

My annoyance with GUI automation isn't a blanket view. The problem is that few teams put the effort into product and test design that successful GUI automation requires. I think GUI automation *can* be awesome – but we (as an industry) don't seem to care enough…yet.

In the Middle

Should you automate everything, or nothing? Should you test everything, or nothing? How about leadership – should you dictate every detail of what your team should do, or give them no guidance at all. The answer for all of these questions – as you'd expect, is "somewhere in the middle".

In my experience, most people handle the "how much" question when dealing with a range of potential solutions by starting with a reasonable mid-point and working from there – e.g. "let's automate half of our tests", or, "We'll test what's most important", or, "I'll give my team some guidance, and then give them some freedom in how they deal with the details."

Those options are reasonable, so the technique seems to work. It, in fact, does work – but I think it can be better.

A brainstorming technique I use (someone please tell me if I've inadvertently stolen the concept) is to first spend a reasonable amount of time focusing on the extremes – because often, some great ideas for "the middle" comes out of that brainstorming. Think, for example, what you'd do if you tested everything (yes, I know, impossible, but think about it. You'd likely use an army of vendors, need some sort of coverage metrics, etc. Then think about what you'd do if you tested nothing (you may have developers own unit and functional tests, and would rely on customer feedback for scenarios, etc.). In the end, there may be something you take from both brainstorming sessions when you figure out what "the middle" looks like.

How about we try another example. Let's say you are testing application compatibility with version 2.0 of the "Weasel" operating system. There were 100 applications written for Weasel 1.0, and you have copies of all of them. How many of those applications do you test? If you go straight to the middle (which, again, isn't a bad choice), you'd probably prioritize the apps by sales numbers

and test the top n number of apps based on how much time you have. Not a bad solution, and one I'd feel comfortable bringing to the team leaders.

But let's think about the extremes for a bit. What would testing all 100 applications look like? We'd definitely need to outsource the testing – but in order to do that, we'd need some clear directions on what "testing" an application entailed. We could write separate notes for each application, but there are probably come up with something generic (install, uninstall, copy/paste, print, major features, etc.) that could work. This solution is certainly going to be too expensive for Weasel management to approve, but we're just brainstorming.

Now think for a while what it would be like to test none of the apps (and not piss off customers). Well, if none of the programming interfaces used by the apps changed, they'd all probably still work. But this is Weasel 2.0, so of course we're going to tweak the APIs. So, maybe it's possible to profile the APIs used by the Weasel 1.0 apps and diff that against the APIs we're changing in Weasel 2.0 and then develop an API test suite that ensured API compatibility. There may be something here…

Of course, neither of these solutions is the right answer (nor are my brainstorming sessions complete), But I'll bet that if you try this approach the next time you're dealing with a range of possible solutions, you'll come up with some new ideas on what you may choose to do "in the middle".

Test Design for Automation

I've been pondering test automation recently. I've complained about misguided test automation efforts before, but it's more than that too. For every tester that cries out that 100% automation is the only way to test software, someone else is simultaneously stating that only a human (with eyes and a brain) can adequately test software.

The answer, of course, is in the middle (see previous chapter).

But I worry that even for those who have figured out that this isn't an all or nothing proposition, many testers have no idea at all how to *design* an automated test – which means that they don't know how to design a test in the first place. The problem I see most often is in the separation of automated and human testing. When approaching a testing problem, you have failed if your first approach is to think about how you're going to (or not going to) automate. The first step – and most important – is to think how you're going to *test*. From that initial test design effort, you can deduce what aspects of testing could be accomplished more efficiently with automation (and without).

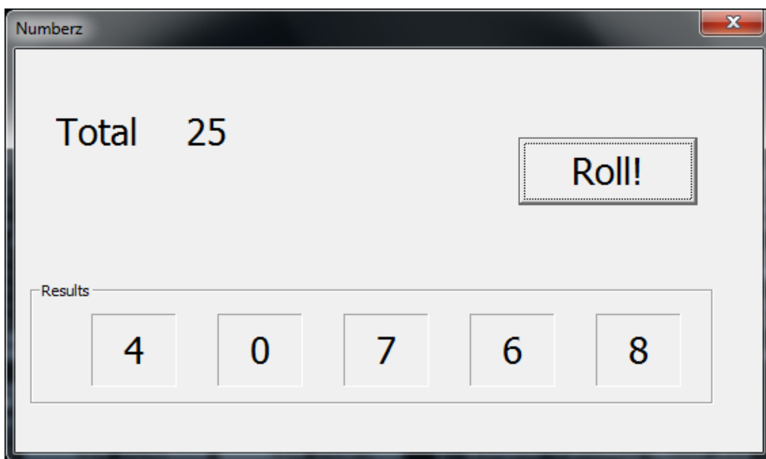
A common variation of this is the automation of manual tests. It pains me to hear that some testers design scripted test cases, and then automate those actions. This tells me two things: the manual test cases suck, and the automation (probably) sucks. A good human brain-engaged test case never makes a good automated test case, and automating a scripted scenario is rarely a good automated test (although *occasionally*, it's a start). Some teams even separate the test "writers" from the "automators" – which, to me, is a perfect recipe for crap automation.

An example would be good here. Imagine an application with the following requirements / attributes:

- The application has a "Roll" button that, when executed, generates five instances of random numbers between 0-9

(inclusive)
- The application totals the output from the 5 random numbers and displays them in the "Total" field.
- There are no user editable fields in the application

For those of you with no imagination, this is how it could look. You can read more about the Numberz application in the *Exploring Test Automation* chapter.



Numberz

From a manual only perspective, layout, user experience, and interaction are definitely areas that need to be investigated, and that are usually best done manually. If I were to write a few scripted manual test cases for this (not that I would), they may look something like this:

Test Case 1

1. Press Roll button
2. Count (use calculator or an abacas if necessary) to verify that the value in the Total field matches the sum of the values below

Test Case 2

1. Press Roll
2. Ensure that the values in the lower fields are within 0-9 inclusively
3. Repeat at least *n* times

Test Case 3

1. Press Roll
2. Ensure that the value in the top section is between 0 and 45
3. Repeat

I have two complaints about the above tests. The first is that executing them manually is about as exciting as watching a banana rot, and the second is that (as I predicted), they're not very good automated tests either.

When designing tests, it's important to think about how automation can (or won't) make the testing more efficient. What I hope you've realized already (and if not, please take a moment to think about the huge testing problem we haven't talked about yet with this application), is that we've done nothing to test for randomness or distribution.

Testing randomness is fun because it's harder than most people bother to think about. We don't have to get it perfect for this example, but let's at least think about it. In addition to the above test cases (granted, the third test case above may be redundant with the first), we need to think about distribution of values within the bottom five boxes. Given the functional goal we're shooting for, we can probably hit all those test cases and more in a reasonably designed automated test.

Pseudo-code: > Loop 100,000 (or more) iterations
> Press Roll Button

> Verify that sum of output and total field are identical
> Verify that output values are between 0 & 9
> Store count of values from output boxes 1-5
> End Loop

Examine distribution of numbers
Examine sequence of numbers
Other pattern matching activities as needed‘

The first loop takes care of the main functionality testing, but where automation really helps here is in the analysis of the output. I’d expect a statistically accurate representation of values, repeated values, and sequences.

That’s stuff you want to use automation to solve. Yet, I keep discovering stories of testers who either don’t bother to test stuff like that at all, or put their automation effort into trying to navigate the subtleties of UI testing.

I don’t’ care whether you’re an automator, a tester, a coder, or a cuttlefish – your goal in testing is not to automate everything, nor is it to validate the user experience through brain-engaged analysis. Your job is to use the most appropriate set of tools and test ideas to carry out your testing mission. You can’t do that when you decide *how* you’re going to test before you start thinking about *what* you’re going to test.

Notes:
I don’t know why I picked 100k for the loop – it may be overkill, but it’s also something you have an *option* of doing if you’re automating the test. I suppose you could do that manually, but you will go insane…or make a mistake…or both.

There’s more to this concept, and follow ups…maybe. The big point is that I worry that people think that coded tests replace human tests, when they really *enhance* human testing – and also that human testers fail to see where coded tests can help them improve their testing. I think this is a huge problem – and that it’s one of

those problems that sounds bad, but is actually much, much worse than that…but I don’t know how to get that point across.

I should also point out that given my loathing of GUI automation, that I’d really, really, hope that the pseudo-code I scratched out above could be accomplished by manipulating a model or an object model or the underlying logic directly. I want my logic/functional tests to test the logic and functionality of the application – and not a flaky UI.

Orchestrating Test Automation

I've been gathering some information on test automation systems recently and will probably have a flurry of posts upcoming on automation and related subjects.

Some observations as I browse what Bing has to tell me about the subject of Test Automation:

- Wikipedia tells me (or once told me) that test automation "commonly involves automating a manual process already in place"
- The bulk of the test automation articles are about UI automation
- There are some reasonably good articles warning about the problems one may run into in test automation projects. None of these articles provide solutions or alternatives.

There's more in my search results, but those three results alone say a lot about what's wrong with the way (most?) teams approach test automation.

I expect I'll write more eventually on all of these points, but my anecdotal experience, along with a few dozen web pages and articles tells me that there's a lot of confusion in regards to test automation.

You see, test design (including the design of how test automation will execute) is really, really hard. It's hard to write sustainable and reliable tests, and it's hard to write trustworthy tests. Double the effort required if those tests involve UI automation. Designing good tests is one of the hardest tasks in software development. That's worth saying again.

Designing good tests is one of the hardest tasks in software development.

Compared to everything else in the equation, the "writing code" part of test automation is easy. Massively easy. Almost brain-dead easy. When I compare writing code to composing music, I talk about the creative and challenging aspect of melody, creating a texture and mood, and figuring out where notes and space between notes help with both. There's also the rote part of the job – chord voicings, doubling, and other parts of filling in the score. The melody and texture choices are the test design of music composition – filling in the rest of the parts is the coding part. Sure, there's creativity in a counter melody, as much as there's creativity in a cool algorithm, but it's still the rote part of the activity.

I fear that testers are worrying too much about the effort and skills required to automate tests – and not enough about what it takes to design reliable, portable, accurate, and trustworthy tests that actually matter.

LOL — UR AUTOMASHUN SUCKZ!

There's a pretty good presentation from the folks at Electric Cloud making the rounds on "why your automation isn't" (and other variations on that title). The premise is, that testers spend too much time babysitting (supposedly) automated tests, and that testers end up doing a lot of manual work in order to keep their automation running. I know nothing about their products, but they have some reasonably good ideas and apparently some tooling to help.

But it's not enough. Crappy automation and crappy automation systems is an insanely huge problem. But we tend to ignore it because – hey – we're running lots of automation, and that *must* be good. To be clear,I have nothing against test automation (although I remain leery of a lot of GUI automation. Done well, it absolutely aids any testing effort. The other 99% of the time I bet it's actually slowing you (and your team) down. Please, please, get it through your head that your job as a tester is to test software and not to create the largest test suite known to humankind.

…and here are some ideas to help you get to automation that ***doesn't suck***.

The obvious place to start is with your code. Do you treat your test code like production code? Do you do code reviews? Do you run static analysis tools? Do you step through the tests with the debugger to ensure they are doing what you *think* they are? Do you *trust* the results from your tests – i.e. if a test fails, are you confident that there's a product bug? If you're spending hours every day grepping through test results and system logs to try to figure out what happened, your automation sucks.

Now think about *how* your tests are run. Are they automatically built for you every day (or more frequently), and distributed to a test bed of waiting (physical and virtual) machines? Or do you walk around a lab full of computers manually typing in command lines? Or do you just run all of your automation on the spare machine in

your office, then upload your results to some spreadsheet on a share where nobody can ever find it. In other words, do your tests execute automatically…or do they suck?

What about failures? Do you look at your automation failures in the morning, gather up some supporting data, then enter a bug? Or are bugs entered automatically – including additional information like logs, call stacks, screen shots, trace information, and other relevant information *automatically?* When a bug is resolved as fixed, do you go run that failing test *manually* to ensure that the bug was fixed – or does your automation system automatically take care of manual (and mundane) tasks such as this for you? How about reporting – how do you generate the all important "Test Result Report"? Does your *automation* system take care of it, or is it a largely manual task.

Do you really have automation – or just a suck-filled wrapper around some suck-infested tests? It's ok, be honest. It's your choice what to do, but, but I bet the maturity of *how* you test software certainly has much more to do with end user quality than the number of tests you have (mostly because the latter metric is useless, but you get the point).

I don't have a solution to sell you – but I can give you an architecture for and end to end true automation system for free. The chapter I wrote for Beautiful Testing[6]. It's all yours – read it and let me know if you have comments – or read it and delete it to free up some disk space. Regardless of my chapter, I still recommend you buy the book – like the other authors, I don't make any money, but the proceeds go to buy mosquito nets to prevent malaria in Africa. More importantly, the book is friggin' cool and I think every tester should own it.

KTHXBBYE

[6]http://angryweasel.com/Articles/Beautiful_Testing_chapter8.pdf

Exploring Test Automation

I try to read a lot about testing in blogs, articles, books etc. A few days ago (ed. A few years ago, at this point), I came across this quote, and it struck me in an odd way.

> "Commonly, test automation involves automating a manual process already in place that uses a formalized testing process"

The source doesn't matter, as it turns out that sentence was copied directly from the Wikipedia article on Test Automation[7]. I've been at Microsoft for a long time now, and although I do try to stay connected with testing outside of the Borg, sometimes I notice that *my* view of a subject is quite different than what's "commonly" understood.

Or, maybe not – but let me try to explain my concerns here a bit more.

I'm all for saving time and money, but I have concerns with an automation approach based entirely (or largely) on automating a bunch of manual tests. Good test design considers manual and computer assisted testing as two different attributes – *not sequential tasks*. That concept is such an ingrained approach to me (and the testers I get to work with), that the idea of a write tests->run tests manually->automate those tests seems fundamentally broken. I know there are companies that have separate test automation team that do exactly this, and I think it's a horrible approach.

Let's use the Numberz Challenge as an example (see sidebar for details if you're not near an Internet connection).

---

[7] http://en.wikipedia.org/wiki/Test_automation

The Numberz Challenge

When you press the “Roll!” button, the app generates 5 random numbers between 0 & 9 (inclusive), and sums the numbers in the “Total” above. The stakeholder’s primary objectives are that the numbers are random, and that the summing function is correct.

To be honest, this is an app where you could probably prove functional correctness via code review, but let’s stick with black box testing for this exercise. The hints in the original post may or may not provide clues to defects in this implementation (that may or may not exist).

Here’s the challenge

If you want to play a little game for me and do some testing, test this app and tell me if it’s ready to ship (according to the stakeholder expectations).

In a perfect world, I’d have an object model or some other way to test the functionality without using the GUI. Sorry - for this example, the world isn’t perfect.

If you want to check it out yourself, a zip file with the binary is here[a].

[a] ibid

Now, to test it!

I know right away that I’m going to perform some manual tests (e.g. make sure the app can launch and close (original version had a bug here), verify look and feel, color choices, etc.). These sorts of tests could be automated, but I generally don’t for a few specific reasons.

- The oracle for look and feel is difficult. Automated testing based on comparing screen colors or app dimensions is fragile. A pair of eyes for a few seconds every once in a

while is much cheaper. *There are, of course, exceptions, and if you're convinced you can pull this off, just make sure you calculate the time spent investigating failures (and potential false positives) into your ROI calculation. I know many teams who have successful UI automation systems, but many more who have a bunch of test auto-crapation.*

- Most systems I work on – and certainly everything that has a UI, gets at least a tiny bit of usage from the product team or a small pilot group before rolling out widely. If the screen is pink, or the font somehow changed to Comic Sans, it will be noticed without the need for an automated test.

When I look at something like Numberz, I also see where I need to use the power of a computer to answer some of the questions I have. For example, I know I need a large enough data set to be confident of the random algorithm, and I know I need extensive pattern matching to ensure that the next number is never predictable. Doing this manually is impossible (or a waste of time depending on what you try to do).

Now I imagine this scenario in the write-test-then-automate-it workflow.

1: Test Case Steps:

```
1) Launch App
2) Press the Roll! Button
```

2: Verify:

```
1) Ensure that none of the numbers is less than 0 or gr\
eater than 9
2) Ensure that the total field is correct
3) Repeat this test at least 10 times
```

Now, the "automator" comes along and writes this test:

1: Loop 10 times
2: App.Launch ("numberz.exe")
3: Button.Click("Roll")
4: // Validate numbers are within range
5: // Validate correct total
6: End Loop

From what little context I have from the interwebs, this appears to be a common scenario — but we've missed the critical aspects of testing this app (testing randomness / predictability). *What we haven't done at all in this situation is **test design**. To me, test design is far more holistic than thinking through a few sets of user tasks. You need to ask, "what's really going on here?", and "what do we really need to know?". Automating a bunch of user tasks rarely answers those questions. (Note – it does answer _some* questions, so if you have a good system and tests you can trust, by all means don't stop what you're doing).

Where I think I have my big disconnect is in the definition of test automation. When I think of test automation, I don't think of automating user tasks. I think, "How can I use the power of a computer to answer the testing questions I have?", "How can I use the power of a computer to help me discover what I don't know I don't know?", and "What scenarios do I need to investigate where using a computer is the only practical solution?".

Perhaps test automation is purely the automation of manual tasks, and I'm attempting to overload the word. I know some folks prefer the term "computer assisted testing", and I suppose that's fine too.

To me (and I'm sure I've used this line before), *it's just testing*. But please **stop** thinking of test automation as the step that follows test design, and start thinking of test design first.

Musings on Test Design

I'm still musing / fuming on the wikipedia quote from the last chapter (note, that as of this writing, the article has been updated and no longer contains that text).

I can't decide whether it bothers me because it's wrong, or because so many people believe the statement is true. In fact, I know the approach in some organizations is to "design" a huge list of manual tests based on written and non-written requirements, and then either try to automate those tests, or pass the tests off to another team (or company) to automate.

This is an awful approach to test design. It's not holistic. It's short-sighted. It's immature, and it's unprofessional. It's flat-out crummy testing. I frequently say, "You should automate 100% of the tests that should be automated". Let me put it another way to be clear:

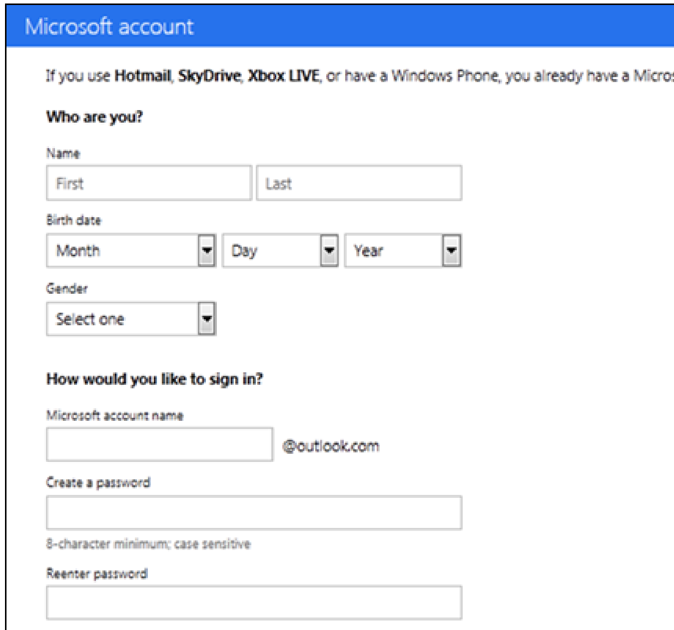
Some tests can only be run via some sort of test automation. **Some tests can only be done via human interaction**.

That part (should be) obvious. Here's the part I don't think many people get:

You can't effectively think about automated testing separately from human testing.

Test Design answers the question, "How are we going to test this?" The answer to that question will help you decide where automation can help (and where it won't).

Here's a screen shot of part of the registration form for outlook.com (disclaimer – I have no idea how this was tested).



Outlook.com registration form

Let's look at two different ways of answering the "How will we test this?" question.

The "automator" may look at this and think the following.

- I'll build a table of first and last names and use those for test accounts
- I'll try every combination of Days, Months, and Years for Birthdate
- I'll generate a bunch of different test account names
- I'll create a password for each account
- Once I try all of the combinations, I can sign off on this form
- (or, they may think, "I wonder what sort of test script the test team will ask me to automate"

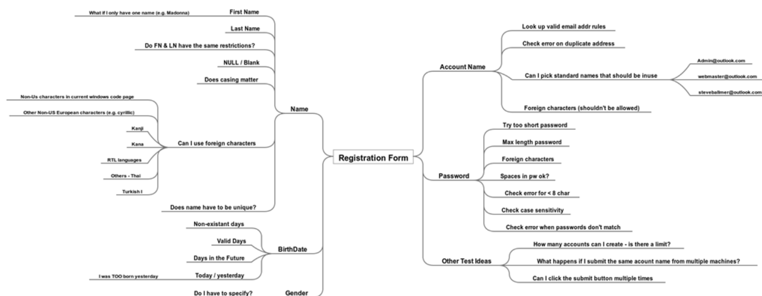
The "human" may look at the same form and think this:

- I'll want to try short names, long names, and blank names
- I'll see if I can find invalid dates (e.g. Feb 29 in a non-leap year)
- Some characters are invalid for email names – I'll try to find some of those
- I'll make sure the 8-character minimum and case sensitivity is enforced
- Oh – I'll try that passwords with foreign characters too.
- Once I go through all of that, and anything else I discover, I can sign off on this form.

I'll fire off a disclaimer now, because I've probably annoyed both "automators", and "humans" with the generalizations above. I know there's overlap. My argument is that there should be more.

In my contrived examples above, the "automator" is answering the question, "What can I automate?", and the "human is answering the question, "What can I explore or discover?". Neither is directly answering the question, "how are we going to test this?".

I could just merge the lists, but that's not exactly right. Let's throw away humans and coders for a minute and see if we can use a mind map to get the whole picture together. Here's what I came up with giving myself a limit of 10 minutes. There's likely something missing, but it's a starting point.



Sample Mind Map

Now (and at no time before now), I can begin to think about where automation may help me. My goal isn't to automate everything, it's to automate what makes the most sense to automate. Things like submitting the form from multiple machines at once, or cycling through tables of data make sense to automate early. Other items, like checking for non-existent days or checking max length of a name are nice exploratory tasks. And then, there are ideas like foreign characters in the name, or trying RTL languages that may *start* as an exploratory test, but may lead to ideas for automation.

The point is, and this is worth repeating so often, is that thinking of test automation and "human" testing as separate activities is one of the biggest sins I see in software testing today. It's not only ineffective, but I think this sort of thinking is a detriment to the advancement of software testing.

In my world, there are no such things as automated testing, exploratory testing, manual testing, etc.

There is only testing.

Beyond Regression Tests

In a recent talk on test design[8] (slides are here[9], I discussed the concept of "useful tests". In my definition, useful tests are tests that provide new information. Almost every test is useful…once – typically the first time it's run where it shows that the underlying functionality is working. From that point on, many tests function primarily as regression tests. We run these tests as a safeguard to ensure that we don't break anything with future changes. They're great for this purpose – and when they find a bug, they are wonderfully useful. But 99% of the time (according to my definition) they are not useful – they don't provide new information (they do provide information – the information that nothing has changed, but it's not *new* information).

To be fair, these are exactly the types of tests I want for unit tests (and probably acceptance tests too, depending on the definition). I love using unit tests to give everyone on the team confidence that refactoring or feature additions do not break basic functionality – but unit tests (and acceptance tests) are usually just a small portion of the overall testing effort.

I should stop here to clarify. On some teams who "do automation", the unit and acceptance test automation is all they do. In this case, the regression only focus *may* be appropriate.

Fortunately, the teams I work on (and with) use automation for far more than regression testing – , we need tests that can be "useful" (provide new information) way more frequently than when something that was working once stops working. Test automation isn't just for regression testing It cannot (and shouldn't try to) replace human brain-engaged testing, but it can certainly do more than automate a bunch of rote tasks. **In my opinion, an automation strategy that only performs regression testing is short-sighted**

[8]http://www.softwaretestpro.com/Item/5060/Better-Test-Design-for-Everyone/Testing

[9]http://www.slideshare.net/AngryWeasel/test-design-for-everyone

and incomplete.

So – how do you write an automated test that provides new information more frequently? A big advantage of human testing is what I call the "I wonder…" principle. For example, "I wonder what will happen if I cancel this operation in the middle?" or "I wonder if this will work if I pull out the network cable?". Computer programs aren't very good at wondering, but they are good at brute force. Model-based testing (MBT) is *a potential* solution here, as you can use it to traverse decision points in the application (randomly, or by applying a graph traversal technique). Traversing an application in unexpected ways often results in finding new issues. Same test, different bugs – and that seems pretty useful to me. Some testers, for whatever reason, seem to be a bit afraid or skeptical of MBT. I think in many cases, potential adopters of MBT either try to do too much with it before they understand the concepts, or think that it's a replacement for some other types of testing. MBT is just another test design technique – and like other test design techniques, it works great in particular situations, and not so well in others.

But MBT isn't the only way to get tests to provide different information on subsequent runs. Simply introducing some randomness to a test can often turn up new issues. For example if you have a "suite" of tests that always run in the same order, try running them in a random order on each run (note – log the random seed so you can reproduce any discovered issues later). Or, say you have a test that does a particular operation five times. Instead of running the operation five times on every test, what about running the operation [between 1 & 10 times] randomly. **Regression tests, for all the value they bring, tend to train the system to pass**. By mixing things up, we often find new issues.

I'm also a fan of data driven testing. Get the test data out of the automation code and into a data file (use XML or another DSL as appropriate). Then, randomize the data used by the test (and automate the randomization too). Then when you're testing a web

page where users enter address information, you can easily mix up a variety of name, address, zip code, etc. data (especially including invalid data), and see what happens. If you send the same data to the system every time, it will certainly pass all of your tests eventually, but I'm almost certain that you'll still miss issues too.

Depending on what you're testing, I'm sure I can think of other ideas that may make your automated tests useful more often. There's nothing wrong with regression testing, but if that comprises all of your test automation effort, you're probably leaving some cards (and bugs) on the table.

Testing with Code

In the previous chapter (Beyond Regression Tests), I mentioned the "I wonder…" principle with the example of "I wonder if this will work if I pull out the network cable?" An area where the often misconstrued "programmer-tester" can put coding skills to work is to automate the "I wonder" part of the test. If removing a network cable during a particular transaction is interesting, then removing it during almost any network transaction would be interesting. So why not add functionality to the test automation system that throttles or interrupts network activity on all transactions – (or make it a configurable variation so you have some test runs that work with consistent network connections).

To be clear, I'm not suggesting replacing exploratory testing with automation. What I'm saying is that many things we do when exploring can be *scaled* using automation.

Fault injection is a pretty typical way for coder-testers to add value to automation (frankly, I don't consider people who just write point A to point B scripted automation to be testers, but that's something I suppose I'll tackle in the comments section). Testing for application behavior with low disk space, low system memory, and any other common failure scenarios is a great way to make your stagnant automated tests find new issues.

Fuzzing is another example. While not *necessarily* a way to make generic automation find more issues, fuzzing is another place where writing automation is the only (practical) solution. Fuzzing, in a nutshell, is the act of intelligently munging any sort of data used by the system under test (note – for a full definition of fuzzing, see the Wikipedia article on the topic[10]. File fuzzing, for example, is the process of taking a valid file, and then mucking with a bit of the internal structure in order to expose problems (including potential security problems) in the way the application handles that file.

[10]http://en.wikipedia.org/wiki/Fuzz_testing

(Note that what I call "dumb-fuzzing, you can take any old file, mess with it randomly, and see what happens – it's a reasonable test, but typically won't find as many security-class issues). A binary file type (e.g. a word doc) typically includes information about the file, including the number of sections in the file and the length of those sections). If you're careful about how you tweak some of the bits in the headers and sections, you can potentially trick the application into crashing – or, if you're really good, into running arbitrary bits of code. You can do the same thing with any sort of binary data – for example, we do a lot of protocol fuzzing – mucking with bits to make sure the receiving end of the protocol does the right thing.

In order to cover all of the permutations, a good fuzzing approach may require thousands of files. Editing ten thousand files in a hex editor isn't my idea of excitement, so automated fuzzing is a good solution. Oh – and loading all those files into an application doesn't seem that exciting to me either, so that part should probably be automated too.

There's more stuff for the tester-coder to do. In a client server org like I work in (note, at the time of this writing, I worked on Microsoft Lync), we have to test against multiple network topologies. Luckily, with a bit of programming, we can do that all automatically and save a bunch of time. Now – to be fair, there is some tax to having all of this infrastructure. With automated tests and fault injection and topologies and all of the other parts of the system moving, a reasonable amount of time goes to maintenance, and that part sucks, but for the most part, a team of testers with programming skills / have the potential/ to do a reasonably good job testing fairly complex software.

However – if your team only writes regression-ish automation – or if your view of the tester-developer is someone who only writes automated regression tests, you are completely missing the point of coding skills for testers. I know (and fear) that this is the case for way-too-many people in the profession today, and I find it

a bit disheartening. I do hope, however, that the next generation of testers will have the big-picture thinking we need to find the harmony between brain and skills that testing needs to grow.

More on Test Design

I had a few goals in the presentation I gave on Test Design (see *Beyond Regressions Tests* chapter for background, links, etc.). One I stated clearly – this presentation was primarily about the *how* of Test Design – not the *what*. This means that I didn't share a big list of test design ideas – in fact, I don't think I shared any unless they helped make a point.

Other points I left (purposely) up to the imagination. I have an inkling that some testers think that there's a master list of test design ideas somewhere, and once they have the list, they'll be up for any testing challenge.

This, of course, is a horribly broken idea.

First of all, Test Design ideas (I've explored, and continue to explore the pattern metaphor with these ideas) are *endless*. Like infinity, as many testing ideas as you can list, I can think of one more.

The bigger challenge is knowing when to use which testing idea – and to me, that's the bigger challenge in test design. Let's say that I'm an expert in Boundary Value Analysis. You can stop laughing now – all that means is that I'm a hotshot at finding boundary issues. The problem is that only a small (but significant) percentage of testing actually deals with boundaries. I may try testing boundaries in a lot of places, but most of the time that effort isn't doing anything for me.

As a tester, you need to recognize the problem your facing, then pick the best test design ideas for that problem.

Here's a real example. It's no secret that I like model-based testing. Sometimes, testers will ask me "how would I use model-based testing in this scenario?". That's a nice question, *but it's the wrong question.* To be a good test designer, you need to look at the problem first (I need to test this application), analyze how the program works, then pick the best approaches. If you start with the approach

and try to make it work, you're probably going to fail. With this example, a tester with MBT in their "tester toolbox" may recognize a portion of the application that 1)behaves like a finite state machine, and 2) recognizes risk in traversing that state machine. *Then* they apply the technique.

The point is that to be a good test designer (and tester), you need a lot of testing ideas, and you need to know how and when to apply them.

And that's really, really hard – and as you know, that's also why it's really, really fun.

Happy Testing

Coding, Testing, and the A word

I frequently go days at work without writing code. On many days, I do "typical" test activities (seeing what works, what doesn't work, etc.) Some days I'm debugging all day. Other days, I'm sifting through the perl and batch scripts that bolt our build process together. On other days, I'm managing projects, tweaking excel, or trying not to cause too many problems in SQL. This afternoon, at least after lunch, was coding time – and probably the first extended code time I've had in a few weeks.

Without too many boring details, I had a backlog item to write a service to automatically launch a memory analysis tool and periodically take snapshots of memory allocations in specified processes. The memory analysis tools (which I also adapted from another project) work really well, but need a lot of manual intervention in order to get accurate data. I wrote a quick "spec" on my white board, wrote tests as I went along, and in about four hours I had it up and running successfully. I'm confident that the service will make it much easier for our team to find and isolate memory issues. To me, writing this tool was a *testing activity*. It doesn't test anything, but certainly makes testing of the product much easier.

On my drive home, I was pondering the universe (and my day) and remembered that there's a sad bit of truth in the software world. It's a fallacy that causes great confusion. It promotes poor strategies, bad hiring decisions, and endless confusion among management, contributors, and consultants. It may very well be the most misunderstood word in the world of software. It's the "A" word that is practically ruining testing.

Automation.

Yes – as much as I hate them, I used a one word paragraph. I did it because the abuse and misuse of "automation" in testing is causing too many smart people to do too many dumb things. That said, I suppose an explanation is in order.

Once upon a time, someone noticed that testers who knew how to code (or coders who knew how to test) were doing some wicked-cool testing. They had tools and stress tests and monitoring and analysis and they found great bugs that nobody else could find – "Test Developers" were all the rage…so we (the industry) hired more of them. Not much later (and possibly simultaneously), someone else decided that testers who could code could *automate* all of the tasks that the silly testers who didn't code *used* to do. Instead of finding really cool bugs, coding testers were now *automating* a bunch of steps previously done manually. Not to be left out, UI automation frameworks began to sprout, with a promise that "anyone can automate – *even a tester*." Now, test teams can spend the bulk of their time maintaining and debugging tests that *never should have been automated in the first place.*

And that was the first step into darkness.

This mindset, although common, kills me:

- Testers who code write automation
- Automation is primarily made up of a suite of user tasks
- Automation is primarily done through the GUI

All of these items are, in my opinion, idiotic - and likely **harmful** to software quality. It focuses test teams in the wrong place and is, frankly, insulting to testers who know coding.

If you're a tester who writes code (or a coder who cares about testing), try this:

- Write tools that help you find important bugs quickly
- Write a few high level tests that ensure user actions "work", but leave most of it for exploratory testing, user testing (assuming you have great monitoring tools), or both.

- If you must write GUI tests for the above, write fewer of them (that way you'll spend less time investigating their flaky behavior).

If you're a tester who writes code, you're not a test automator, you're not a test automation engineer, and you don't write test automation or test code.

You write code. And you test.

You write code that helps the product get out the door sooner. And if the code you write (or the testing you do) doesn't help the product, why are you doing it in the first place?

Last word on the A word

Some have asked if I'm against all automation. My tautological answer is that I'm against *bad automation*. My better answer is, no – I'm not against automating tasks for *some* testing. Automation is overused and overvalued in software – while the coding of diagnostic and analysis tools is extremely *under*valued. In many cases quick check of basics, some unit tests, monkey testing, etc.), automating a set of user tasks is beneficial and provides a lot of bang for the buck, but focusing entirely on automation of user tasks is a waste of any programmers time.

I wrote a chapter in a test automation book (Experiences in Test Automation) a few years ago, but on principle, I didn't write about automation. I wrote about a simulation framework that made it easier to write *some targeted automation*. To me, writing something like this is a great use of a tester's time – often much better than to write a suite of automated scripts.

The point is, if your testing strategy is *entirely* to automate a suite of user tasks, you are doing way more to keep "automators" busy than you are actually providing valuable testing. We can debate the goal of the testing activity, but one thing it is **not**,is the activity of running through a bunch of user tasks over and over. In fact, one of the touted benefits of automation is repeatability – but no user executes the same tasks over and over the exact same way, so writing a bunch of automated tasks to do the the same is often silly.

I think I'm done writing about automation for now. I may continue to write about code or debugging or testing. My advice for testers is to worry less about "automation". Ignore it if you can – but at the very least realize how overloaded…and overblown the word is is the software community and take a holistic approach to software testing.

Resources

My blog, if you haven't figured it out, is at http://angryweasel.com/blog. You can also find me on twitter (@alanpage)

A link to my chapter on Large-Scale Test Automation (from *Beautiful Testing*) is here: http://angryweasel.com/Articles/Beautiful_-Testing_chapter8.pdf

You can find a list of all my published works at http://angryweasel.com/blog/?page_-id=381

Thanks again for reading.

AW