Lightsabers, Time Machines and Other Automation Heuristics

Heuristics are used in testing as a rule-of-thumbs or prompts for solving a particular problem or class of problems. One important characteristic of heuristics is that they are fallible in certain situations; though are correct in most situations. And then there are anti-heuristics. They are fallible in most situations.

Automation, like all of testing, is an inherently heuristic activity. Some of the risks associated with it can be managed by making conscious decisions around which heuristics you want to employ and which anti-heuristics you want to avoid.

This paper, and the companion talk, identifies and names a handful of heuristics that apply to automation. While not an exhaustive set, it is a useful one and will set the reader on the path to identifying and collecting their own set of automation heuristics.

Lightsabers

In the Star Wars stories, the lightsaber is the physical tool of the Jedi when their primary tool, the Force, is insufficient for the task at hand. In testing, we use automation when our primary tool, our brain, is insufficient for the task.

An important step in the apprenticeship of a young Jedi is the creation of their first lightsaber[1]. This happens towards the end of their apprenticeship and prior to that they use either a blunted lightsaber or one that is not attuned to themselves and their own energies.

This lack of attenuation is exactly what happens when using ‘generic’ automation tools. It is not the vendor’s intent to be malicious or do harm, but by their very nature they need to support a lot of situations to be viable in the market. That support leads to generic approaches that sometimes means you have to adapt the way your application is built to support your automation tool.

When using a closed-source tool the challenges are even greater as you are beholden on the vendor for any fixes to problems you might discover. And what might be a problem in your context might not be to them so they might be unwilling to fix your problem.

Just as a Jedi building their own lightsaber solves the problem of attenuation and bug fixing for their tool, building your own automation tool or framework using Open Source

---

[1] http://www.amazon.ca/Jedi-Path-Manual-Students-Force/dp/1452102279/

components addresses these problems for your automation. No longer do you need to use generic solutions to specific problems, but you can use one that solve your specific problem precisely. By the same token, if a problem occurs, you have access to the source code and can produce a fix to it immediately.

The Lightsaber heuristic is of course fallible though. A Padawan (apprentice Jedi) does not just wander out on their own and build a Lightsaber; they have a Master who guides them through the process the first time. Often to protect them from themselves. So too with building automation. Someone who does not have experience with a number of different automation tools or techniques could stumble upon something that works, but is more likely to head down a number of false paths, dead ends and end up causing more harm than good.[2]

With Open Source, as with commercial automation, you still need to be aware of the 'Know Where The Sun Is' heuristic when fixing problems you encounter. For example, the Selenium WebDriver API is Open Source and anyone can build custom copies, but if you have a question on the Java, Ruby or Python implementations, the best way to get information is during European business hours. For C# though it is business hours on the east coast of the US. During WinRunner's heyday you tried to call in the (really) early morning North American time so you hit the support center in Israel.

There will be times when technology choices dictate going with a commercial, closed source tool or framework. But teams that really succeed with automation tend to be ones that build their own tools using open source components.

One final thing to remember with lightsabers is that they were wielded for both good and evil purposes.

Time Machines

It is impossible to determine the Return on Investment (ROI) for automation without access to a time traveling device. This is a fairly outrageous claim to make in a world where budget justifications need to be made quarterly, but it really strikes at the core of why we use automation in the first place.

Automation is not a technique to be used to increase some metric in a spreadsheet regarding the number of scripts executed or coverage percentage. Its not really even a technique for doing things faster than humans or that humans are not good at (like boring data input). Automation is just a technique for gathering information. And that information is consumed by humans who make decisions with it.

---

[2] My current automation framework, Saunter (http://element34.ca/products/saunter) is the 5th or 6th incarnation of these ideas and is only now at the point where it is really useful

It is this whole usage in decisions part that is the tricky bit. Consider the following situations

- Based on a series of automation results, a tester decides to investigate a particular part of the product and finds a bug
- Based on a series of automation results, a tester decides to investigate a particular part of the product and does not find a bug
- Based on a series of automation results, a tester decides not to investigate a particular part of the product and misses a bug
- Based on a series of automation results, a tester decides not to investigate a particular part of the product and does not find a bug

Anyone who has worked in / with automation has found themselves in each of these situations. And what was the outcome? And over what time horizon is that outcome being measured across? The only way to do this measurement is to look into the future at every single decision either directly or indirectly made with input from automation and see how it affected the company.

And of course, if you have a time machine at your disposal, your competitors do as well. Which means you have to check the impact of decisions against the information received from automation but also from their reactions to those decisions, and their responses to them, and...

Automation ROI Calculators, such as the one from ATI[3] should of course themselves be considered heuristic (at best; an example of poor metric collection and management at its most cynical). But the problem with them is they do not have an input field for 'future money earned by using automation' or 'future money lost by using automation'. Because both are impossible to measure -- without a time machine.

The whole notion of ROI often comes up as part of the justification process for using expensive, commercial automation platforms. Building your own lightsaber can offset part of the need for a time machine, but a better solution is to approach automation with the realistic understanding that it's purpose is to provide information. Some of that information will of course be useful, but some will not be; just as is the case with manual testing. And exactly as you do with manual testing, when the cost of acquiring that information outweighs its value to the organization, you stop. No time machine required.

The One Ring

Tolkien's books revolved around a magical artifact, the One Ring, which was intended for use by Sauron to control all the peoples of Middle Earth. Inscribed on the ring are two of the more famous lines from the books

---

[3] http://www.automatedtestinginstitute.com/home/index.php?option=com_content&view=article&id=58&Itemid=65

One Ring to rule them all, One Ring to find them,
One Ring to bring them all and in the darkness bind them

With automation, this focus on One is a symptom of homogenous thinking and is an anti-heuristic. That is, it is a bad idea most of the time.

I primarily see two variations of The One Ring when talking with companies.
- Automation language choice
- Automation tooling choice

The automation language choice version is perhaps the most insidious and seems to happen more in companies whose primary development language is Java (which usually also means that they are large companies organizationally). But it can happen in a one person company as well.

The question to ask in order to determine whether The One Ring has started to spread its corrupting influence on the organization is '*why are you using language x for your automation?*'. The answer all too frequently is '*because the application is written in language x*'. And that may be exactly the correct choice, but too often it was made without any consideration
- to whom will be creating the automation
- their skill sets
- and what the automation [really] needs to do

It is not uncommon to see excellent non-programming manual testers 'converted' to being writers of automation only to flail and ultimately fail. This is because automation *is* programming and it is a skill that needs to be learned. Though some languages are just easier to learn (like Python or Ruby).

It is also easy to choose the same language for both the application and automation because there will '*ease of integration and ability to share code*'. But if the integration is through a REST interface then you might have fallen under the sway of The One Ring.

The second variation of this anti-heuristic occurs more in places that have implemented some sort of commercial Application Lifecycle Management System (ALMS) such as Microsoft's or HP's (but sometimes it is an in-house tool (lightsaber) that was built by a team member). These often cost a non-small amount of money to purchase and involve significant workflow disruptions to implement. And having incurred both there is a mandate to use the tool. In this situation it is important to remember that these costs, while non-trivial, are sunk costs -- that is, no matter how much you use the tool they cannot be recovered.

The argument against calling these sunk costs is that by using the tool you are amortizing the costs of it over a larger period of time. Which is correct from a financial reporting perspective, but if you are not getting the information the business requires in a timely fashion you are actually adding *more* cost to the tool by using it.

The solution is of course to throw The One Ring into the fires of Mount Doom where it was forged. But if that isn't convenient, then honestly assessing tooling and language decisions both at the beginning of a project and periodically during it to check that they are still correct will go a long way.

The First Hit Is Free

One of the more trendy 'movements' in software these days is the Lean Startup which has a number of important and very cool ideas teams should consider. But one which it also includes is Continuous Deployment; the process where code travels through a series of automated gates and makes its way to production.

On.

Every.

Commit.

The net effect of this is that new features and bug fixes reach the audience in the shortest possible time, but it also removes the human testing process from the equation. A step I consider unethical. Instead, one should look at Continuous Delivery which uses automation where it is appropriate, and manual steps when they are appropriate. And installing new builds for a functional automation run is absolutely one of those appropriate places.

Consider Continuous Delivery as Continuous Deployment with ethics.

Implementing either is a disruptive event both from a process perspective and a political one and often meets resistance. Automation, especially of the Functional variety (Selenium, Watir, etc.) is a fantastic way to introduce Continuous Delivery into an organization through a back door.

What are Continuous Delivery practices that are required by Functional Automation?

- *Automatic Resource Provisioning* - Both physical and cloud machines need to have all their packages and configurations controlled by some central means. Once this is in place, there should never be a need to log into a machine [to do maintenance].
- *Application Deployment* - The application needs to be able to be installed and configured on the target machine without human interference. This could mean enabling 'silent mode' for installers or scripts for moving files from a build server to the one being configured.
- *Database Management* - All interactions with the database from a management perspective including schema and data changes need to be handled as migrations. This allows for testing of changes before production and removes the risk of type-o's happening on the management console during deployment. As with provisioning, once

this has been implemented there should be no need to every issue any SQL commands into the management console.
- *Continuous Integration* - The CI server becomes the heart and mind of the entire process. Not only does it control the entire build pipeline but it informs the user about past trips along the pipeline.

Depending on the organization, each of these will have a different effect on the efficiency of its operations, but I have found that the Database Management portion actually has a greater effect on perceived application quality than does the execution of the functional automation it is being implemented for.

Once all these pieces are in place and debugged for the automated functional tests, its is not much of a leap to implement them for the manual testing environment(s). And once any new problems are discovered in the process and resolved then you roll it out for pre-production. And then production.

Functional automation that is executed without any intervention is fantastic in reducing the number of 'broken builds' that end up in front of humans. Without having to rely on the Operations team to constantly prepare and maintain the testing machines also means it is more efficient. And it just so happens that the techniques to do this are the same ones that are needed to do automated pushes to production, but by doing it in the early environment the functional automation acts as a gateway drug towards Continuous Delivery.

Shanhaiguan

Shanhaiguan is the city in China where the Great Wall meets the Pacific Ocean and is an anti-heuristic that deals with the most challenging part of automation -- the people writing it.

Walls serve two primary purposes; to keep people out and to keep people in. In too many organization charts they also are used to consolidate control. In each situation, it comes at the expense of communication. And communication between parts of the organization attempting automation is vitally important to the possible success of the effort.

Automation requires the entire team to be successful.
- *Product Owners* - Need to communicate their desires for what the product should do. Without their initial expression of capability there is nothing to compare against to ensure the scripts do what they are supposed to do.
- *Application Developers* - Today's applications are too complex to not to have automation hooks provided by the development team. They may not know which hooks are needed though. Whomever is automating needs to be able to talk to the developer to request these. And have those request not be ignored.

- *Operations* - The configuration and deployment of hardware and software resources and to maintain the build pipeline.
- *Script Developers* - The hub of the automation wheel, they need to be able to talk to anyone in the organization when it relates to getting the automation as effective as possible.

Being able to succinctly place people into these buckets could itself a symptom of other organizational disfunction. From the Agile community there is the Whole Team concept where everyone involved in a product is a member of the product team first and their 'other' role second. The difference between identifying as 'a scripter for project x' and 'a member of project x who happens to script' is subtle, but important. Everything done on a project should be done for the project's well being. Including automation. Not automation for automation sake and the project is irrelevant.

When everyone is working for the betterment of the project as a single team, there tends to be less walls in place between people of various specialties. But even when there are fiefdoms where communication in discouraged by formal processes, informal communication will eventually start to sneak into the system either at the proverbial water cooler or instant messaging or email. One just needs to be aware that those imposing the restrictions will sometimes not look fondly upon them being avoided. So weigh the risks -- and rewards.

If walls, either physical or imaginary are placed between the team members, their ability to communicate is impaired. But like the Great Wall, these walls can be worked around -- it just might be extremely unproductive. And you might get your feet wet.

Switches

Adding the ability to switch something on or off depending on a configuration setting is not a new technique but one that is getting a revival from the DevOps moment as it can decouple production releases from feature availability. But when you start to bring automation into the mix, it is useful to put switches on everything.

Especially third party content.

The primary goal of [functional] automation is to gather information about the current behavior of the application. Too often though, that information gathering is held hostage by a slow or unresponsive third party widget or advertisement. Rather than be a victim it makes sense to flip the switch that disables that widget in the automation environment.

Simply disabling the loading of something might throw off the overall look-and-feel of the application though as the design relies on a certain sized item in a certain location. For situations like that we need a smarter switch. In the case of an ad or widget the server could present to the user a static, local image in place of the expected content. Now the

speed penalty of the third-party content is not paid but the application looks the way the designer intended.

Another type of smart switch is one that replaces the behavior that will end up in production with one that is useful for automation perspectives. Consider the automation required to verify the error messages for a typical payment flow. There are a lot of scenarios that not only require the third-party credit card processor to be available but also presumes you can trigger all the possible responses from within the scripts. Both are pretty big assumptions to be hanging an automation attempt around. A better option would be have a switch that can be thrown that will respond with a predetermined behavior given a specific input (such as a CVV code in the credit card example). With that in place you can easily automate checking of all the error conditions even when the payment provider's system is unavailable.

When doing this sort of switch however it is especially important that your automation does a full verification of the event though. If it does not, you could find yourself pushing to production with the switch thrown[4] and no obvious indicator that key parts of the flow are being short circuited.

Switches are are often also a leading indicator of whether or not an application has been built using TDD techniques. These switches and ability to isolate and control functionality is one of the tricks used to get a high degree of coverage by the unit tests. As a result, applications built via TDD tend to be easier to automate than those that are not.

The Second Noble Truth

One of the core principles of Buddhist philosophy are the Four Noble Truths. The second of these is roughly translated as

The origin of suffering is attachment.

Yet when it comes to automation, attachment is common. And then we wonder why we suffer.

The first type of attachment involves the actual hardware that the automation runs on/against. Configuring machines is a time consuming activity and once it works you don't want to sacrifice the time it takes to re-do it. But this leads to compromises in machine freshness, patch level compliance and quick hacks directly made to the machine. Thankfully, automated configuration management (see *The First Hit Is Free* above) provides relief from that pain, and so from the attachment.

---

[4] And that's not good. Trust me.

The second revolves around functional automation scripts and is far more engrained in teams that do automation.

If the point of testing is to gather information and thus the point of automation is to facilitate the gathering information, then the value of the automation drops significantly when there is no more information to be gathered. One should release themselves from the attachment to the script and delete it from the repository.

With functional automation, the part of the process that generates the most value for the organization is the creation of the script itself -- not the however-many-thousand times it subsequently gets executed. It is during the creation of the script where the author is actively engaged with exploring the system, following data, asking questions to the product and development teams, etc. It is not surprising then that this is when the most bugs are found. And when bug fixes have unit tests created for the fixes then the creation step of a script will likely be the only time a bug is found [by the script].

If a script is unlikely to find provide new information once it has finished being created, why keep running it on every commit? Or at all? The answer is attachment.

Baby Steps

Baby steps get on the bus, baby steps down the aisle, baby steps... - Bob Wiley

In the movie *What About Bob*, Bill Murray's character comes to grips with his OCD by latching onto the mantra of 'Baby Steps' -- taking life in small, manageable slices only as they occur. It is important to approach automation projects with this mantra as well.

Consider the activity of creating Page Objects for an application. It is tempting to go off and create POs for every element and every action on every page and then just glue them together in interesting ways through scripts. This is the very opposite of the Baby Steps approach and pushes out the time when the scripts can produce information[5].

Rather than building everything out at once, create the script and the PO(s) in tandem *as you need them*. By taking this approach you are getting information from the automation immediately. Which is, again, the purpose of this kind of automation. It also means that isolating problems both discovered by and created by the automation much easier as the changes are smaller.

The Baby Step approach also minimizes the amount of re-work necessary when something changes as the PO and the application are in sync from the start.

---

[5] Much in the same way that excessive test documentation prevents actual testing

To some degree, this heuristic brings the to automation the YAGNI[6] and iterative development that have been popularized by Agile to some success.

Don’t Kill Unicorns

There are certain things you really should not do -- like kill unicorns[7]. When creating automation there are lots of unicorns around, but one that I keep seeing is the automation of sites that are not under your [company’s] control.

The scenario this most often comes up in is a registration flow for applications in which an email is sent containing a verification link the user is supposed to click. Automating the first part is simple, as is the last part, but the retrieval of the verification link is where unicorns get involved. And often takes the form of automation GMail or similar sites.

The only people who should automate GMail are those employed by Google and work on the Gmail team. Anyone else is killing unicorns.

There are two ways to avoid killing a unicorn in this particular situation.

- Access the site’s POP3 or IMAP interface directly via your language’s libraries. Both are standardized protocols and removes any interaction with the interface from your scripts. Instead, it is just another service the script is using.
- Reach into the application’s database to retrieve the code and build the link in the script. Your application’s server has no way of knowing if the link originated from a physical click inside an email or a select statement in the code. Nor does it care.

Both these solutions will work, though the second one is preferred as it removes the requirement of the third-party system being available and behaving in the expected way.

Aside from automating third-party services, another unicorn comes in the form of automating things that are more efficient to do manually such as page layout/ content testing[8].

---

[6] You Ain’t Gonna Need It - http://en.wikipedia.org/wiki/You_ain't_gonna_need_it

[7] or make baby pandas sad

[8] Though there are services that can do this for you which is somewhat akin to hiring a Big Game Unicorn Hunter; you get the trophy but don’t get your hands dirty.

Using a Switch to disable third-party items in the application is a good first step to avoid the needless slaughter of magical creatures. Constantly reflecting on whether you should be automating something is however remains the most effective method of preventing wholesale slaughter of unicorns.

If you meet the Buddha

If you meet the Buddha, kill him - Zen Master Linji

This Buddhist koan does not of course mean one should commit literal murder. Instead what it is saying is that if you believe you have a correct image of the Buddha or the nature of Enlightenment then you should kill that idea as it is false.

Killing false ideas is important in all testing, including automation. Recall that both activities are heuristic based. Which means any any ‘best practices’ or ‘expert advice’ is fallible. It is therefore important to constantly look at the path you are on and determine whether the practice is one that you should continue, or should kill. And that includes every single heuristic mentioned in this paper.

But of course, even that is heuristic.