Concepts

Automation

Guidance

People

a fear and loathing of testing

Ash: *We were somewhere around the waterfall, on the edge of the software lifecycle, when the tests began to take hold. I remember saying something like:*

I feel a bit lightheaded. Maybe you should drive.

Suddenly, there was a terrible roar all around us, and the software was full of what looked like huge bugs, all swooping and screeching and diving around the computer, and a voice was screaming:

Holy Jesus. What are these goddamn bugs?

Dr. Gonzo: Did you say something?

Ash: Hm? Never mind. It's your turn to test.

Ash: *No point in mentioning these bugs, I thought. Poor bastard will see them soon enough.*

when should we be doing automated testing?

Automated tests, that are written **before** the code; capture the intention of the code, inform design decisions, provide rapid feedback and let us know when we are done. All of this gets us thinking about testing and ensuring that our code can be automated.

One view of test automation is to write it **after** the system code has been written so the automation has to cope with less change. We've found this view doesn't hold up in practice, firstly we spend a lot of time reverse engineering the code to automate it. Secondly, if the code is changing then this is when we need test automation the most to provide us with a safety net.

Automated tests that are written **after** the code do not directly inform the design nor do they provide rapid feedback. When writing automated tests in this way we need to ask ourselves; why are we taking this approach?

If we are doing it to provide test coverage or run in the CI build then we are coming to the party late. Without visibility into what automation already exists we could be duplicating test effort. If these tests will help us build a better product then they should be written **before** the code.

If we are using automation to do exploratory testing and we intend to throw the automation code away afterwards then we can write the tests **after**. Not all automation needs to be kept it just has to help us explore.

what we do in the case of an automation backlog?

The behaviour people have when they are behind is often more damaging then being behind. We're going to focus on a couple of ways we can climb out of this hole.

Firstly, get the team to help us catch up!

Moving the culture to a test driven approach will fix the backlog and prevent it from happening. As changing culture can be long term, we can use WIP (Work In Progress) limits; an easily implemented tool from Lean, to make the team aware of the problem and to move in the right direction.

For example, once the tester's WIP limit is reached, before anyone can start more work, they need to help the tester finish something. Get the team to stop starting and to start finishing. Optimizing the team as a whole increases work flow more than optimizing individual components.

The second tactic is damage control, and will stop us chasing our tails until the end of time. This approach assumes a team that is unwilling/unable to support the testers. For new work, we prioritise based on risk; covering the low priority work with mostly manual testing to stop the backlog from growing. To deal with the existing backlog, we write smoke tests to give us thin coverage and confidence over the functionality, filling in the gaps based on priority, balanced with the new work coming in.

More on information on WIP limits:
http://leanandkanban.wordpress.com/2009/05/14/wip-and-limits/

testing integration projects

When problems arise (often) in integration projects (all of them) a lot of time and energy is spent arguing who's at fault. This is as about as useful as arguing whose side of the boat has a leak.

It's common for Team Upstream and Team Downstream to test their systems in isolation. However, the problems lurk in international waters, between teams. We must test the integration. Integration testing is like voting, do it early and do it often.

Traditionally, groups are brought together through marriage. In lieu of this, we've had a lot of success sending emissaries to work with other teams. This doesn't mean endless meetings. It means joining forces and working together, reducing the us and them mentality.

We may be told that our responsibility ends with our system boundary, stay out of international waters. We may be told that it will all just work if we build it to spec. The reality is that we need to make sure the entire system works across all teams.

Current thinking is that our brains are geared towards living in small family groups, in competition with others. This distorts our view of other teams, causing us to presume they are either malicious or incompetent. We even dehumanise them, giving them nicknames like Team Downstream, instead of recognising them as fellow people doing the same job as us.

As the elders say: you must test the integration (and take these mushrooms!)

Disposable Automation

In our experience, testers have an unhealthy attachment to automated tests. We're going to talk about times when throwaway automation is really helpful.

Record and playback

Sick and tired of clicking through page after page to find what you want to test? Record your path, run it, and test what actually matters. Record and playback is quick and easy. The code it creates will make your eyes bleed which doesn't matter as long you dump it as soon as you're done with it.

Exploratory Automation

Sometimes you need to test things that can't easily be done manually. We were exploring a bug lurking deep within a server and found ourselves manually crafting HTTP headers in telnet. We realised it's a lot easier to do this in code. So we did. We found the bug and threw the automation away.

Permutations and Combinations

There's an adage that you can't test everything. Sometimes, your tester senses tell you to cover a part of the system thoroughly. This can be done with a script that generates the various combinations. Run it over night and don't leave your number.

Automation you decide to keep, you decide to maintain and "The things you own, end up owning you." Tyler Durden

Do you know about test fatigue?

> Fatigue is a normal result of working, mental stress, overstimulation and understimulation, jet lag or active recreation, depression, and also boredom, disease and lack of sleep.[1]

We could rewrite the above quote to be:

Test fatigue is a normal result of testing, delivery pressures, thrashing, uninteresting work, disenfranchisement, mechanical work, bad practices and working overtime.

What's wrong with that?

> …mental fatigue, in turn, can manifest itself both as somnolence (decreased wakefulness), or just as a general decrease of attention, not necessarily including sleepiness. Decreased attention is known as ego depletion and occurs when the limited 'self regulatory capacity' is depleted. It may also be described as a more or less decreased level of consciousness. In any case, this can be dangerous when performing tasks that require constant concentration, such as driving a vehicle… [or testing][1]

This is a big topic, we have a lot more to say…stay tuned.

[1]: http://en.wikipedia.org/wiki/Fatigue_(medical)

Dealing with test fatigue

Here are the problems we raised in our last post and ways we deal with them.

working overtime - You can't test tired. If you're going to be working overtime for several hours, have a break. Take time away from the project and go out for dinner, like a second lunch. Adjust the workplace to your style, watch YouTube together and take frequent communal breaks.

delivery pressures - The more pressure the team is under, the more likely they are to make mistakes and the more you need to test. DON'T PANIC. The less time you have the more you need to get it right the first time.

thrashing - Make a task list of what needs doing and divvy up the work. Stop people from interrupting (think: cone of silence) by politely explaining the urgency of what you're working on. Remember, prioritisation! It's normally better to finish some things than to partially complete lots of them.

uninteresting work - Spice up boring work by trying it in a new way. Any technique will do, invent your own or try something from your favourite testing blog. You can make work fun.

mechanical work - Automate it, computers love repetitive tasks. Even if you dispose of it later, you're saving time. Delegate it to the development team, they love repetitive tasks.

bad practices & disenfranchisement - Why are you doing this to yourself? Good testers are a rare breed. Other companies want you, we want you. If you can't fix it, leave.

Testing in the deep end

Testers don't have the luxury of a friendly group of people checking our work. Instead we get an angry mob with torches and pitch forks. We have to be honest about our limitations. It's better you reveal them at the beginning, then get bitten by them at the end.

As a rule: If you're testing something that you can't explain to someone else, then you probably don't understand enough to effectively test it.

How do you approach testing something you don't understand?

Find people who understand it better than you. Talk to them. If you can't find real people, find Google. Regardless of your source, cross reference information and consider the author's bias.

There are fundamental techniques in testing that apply to most problems. Use them to explore, break it down and ask good questions. A common testing pattern is:

Given *precondition*
When *action*
Then *expected thing*

Use this to think about what you need to set up, what you need to do and what results you should expect.

Since we can't know everything we're always out of our depth to some extent. So don't panic. We have touched on three techniques to approach this problem: be honest, find someone who can help and fall back on your testing fundamentals.

Sometimes the most intelligent thing you can say is "I don't know".

Chicken Little

Once upon a time there was a tiny chicken [tester] named Chicken Little. One day Chicken Little was scratching in the garden when an acorn fell on her head. “Oh,” cried Chicken Little, “The sky is falling! I must go tell the king [project manager].”

It’s important to remember that this is a tester who really cares, we need to harness their passion. A panicked approach causes stress and real problems get lost in the noise. The tester will lose credibility, become marginalised and burnout.

Harness the passion!

We need to work closely with these testers who are emotionally invested and vulnerable to criticism. How they arrived at this behaviour is irrelevant. Two things we’ve found that help are to teach them prioritisation and to value quality over quantity.

Teach them to prioritise

Ask them to rank bugs in the order they would like them fixed. If they struggle, begin by ranking one critical and one trivial bug. This forces them to understand some bugs are more important than others. Once they’re all ranked, discuss at which point we could release with the remaining bugs.

Value them

Publicly acknowledge them for finding the good bugs. Let them see their good bugs being fixed. Recognise their less important bugs and use their prioritisation to explain why they won’t be fixed.

Teach them how to find the important bugs … coming soon

…and they all lived happily ever after.

Validation and Verification

So what is the difference?

It's simple really, as long as you don't read the plethora of arduous ISO and IEEE standards.

Verification is making sure what we built is working as we intended. This is acceptance testing, executable specifications and exploratory testing. These are testing fundamentals.

Validation is asking the question, did we build the right thing? There is little point in building a entirely configurable system if the owners have no want or need to.

Our definitions, like most of the ones we've found, are defined in the past tense, which illustrates the real problem. Why wait to ask such important questions? It's Madness!

Ask these questions early and often, then you'll never have to bother with these definitions again:

- Are building the right thing?
- Is it working?

Defining your testing as verification or validation is largely irrelevant in the grand scheme of things. Dissecting testing into small chunks is a useful way to learn new things, however in the real world, evolving your expertise to include new things, like user experience design, is more effective.

Mule Testing - proactively testing assumptions

We were building a shiny new system that relied on data from a poorly understood legacy monster. Assumptions about this data were baked into our system. These unchallenged assumptions turned out to be wrong. Our shiny new system was no longer so shiny.

The wider the belief in the assumption, the more it's engrained in the business, the greater the need for it to be tested.

Why trust when you can know?

An example Mule test (*it's blogging by example!*)

Start with the assumption: *"All products must have a category"* Find a way to challenge or validate it. In this case we would run a simple query against production data:

```
SELECT * FROM products WHERE category IS NULL
```

If the assumption holds true then rest easy. If it turns out to be false, congratulations you have just prevented a major bug. Share it with the team and update your old assumption to include the new facts. In this example *"A product does not require a category"*.

Mule testing has limits. It only helps you test assumptions that you know about. If the magic combination of data that breaks your assumptions doesn't yet exist, it won't fail. It only works with access to the latest production data.

Why the name mule testing? Because some people got the wrong idea when we called it ass testing.

Mule Specs - automated assumption testing

In our last post we talked about mule testing. Assumptions need automation because they're the foundation our systems are built upon; they can change at anytime. Mule specs are a way to automate mule tests.

You can use any automated testing tool - the one your project already uses is probably fine. Unless it's QTP. Below is the example from the last post in RSpec using the sequel gem.

```
describe 'products' do
  it 'do not require a category' do
    sql = <<-SQL
      SELECT count(1) as row_count
      FROM product
      WHERE category IS NULL
    SQL

    at_least_one_row_exists sql
  end
end
```

We use two helper functions as we phrase our tests to expect either at least one result or no results.

```
def at_least_one_row_exists sql
  DB[sql][:row_count].should != 0
end

def no_rows_exists sql
  DB[sql][:row_count].should == 0
end
```

Getting production data

Mule tests require prod data, the older and less realistic it is, the less certainty you have in your assumptions. Running Mule Specs on production data doesn't mean running them on production, that's a really bad idea. Copy the data elsewhere before execution. We arranged a sync from production every night and our Mule Specs run against it. So, when we arrive in the morning we know that as of yesterday, all our assumptions are still true.

Mule specs give us more than just a way of verifying assumptions. Written well, with good reporting, you produce verified documentation that's updated every night when the mules run. Get the entire team involved. Ensure the analysts note their assumptions as they go and have the testers and developers implement them.

Follow your heart, run with the mules every night.

Ask better questions - Listen!

Everything that follows is a result of what you see here.

Testing relies heavily on asking questions. Questions allow us to challenge assumptions, confirm what we already know and uncover the unknown. Coming up with the right questions and understanding what to do with the answers is a real skill. Consider yourself a detective or scientist, whichever you find more motivating.

Have you read/seen ‘i Robot’? In the story, Detective Spooner has to solve a murder. It starts with him questioning a hologram of the victim, Dr Lanning. The hologram is a simple program, it can only give limited responses to specific questions.

Detective Spooner collects pieces of information, assesses them and re-evaluates what he knows. He uses that to piece together the right questions, fueling the cycle until he uncovers the fundamental flaw that had made it into (mass) production.

He could have asked the hologram many mindless questions, but that may never have allowed him to reach the right one. Testing can be as simple as asking questions but they are futile if you’re not listening to the answers and constantly evaluating what you know.

Beware of robots.

challenging assumptions

Many bugs can be prevented by challenging assumptions. Challenging the assumptions every one holds as well as paying attention to the seemingly small ones, will yield great results.

Small things can be the most dangerous as they tend to go unnoticed, then gang up on you. It's more common for projects to get overwhelmed by a build up of small things. The devil's in the detail.

To find assumptions, listen to the way people speak. Assumptions can found in sentences that contain:

- must, always, mandatory, required
- impossible, inconceivable, never
- should, ought
- doesn't make sense

Doesn't make sense is a favourite. This planet is filled with humans who do many things that make more money than sense.

An Example!

Consider the following story:

As a customer
I want to know the average activity
So that I can compare this month's activity against the average

Sounds simple… but if you look a little deeper:

- What do we mean by 'know'?
- What do we mean by average? Geometric, harmonic or arithmetic mean?
- What activity?
- What do you mean by month? How many days are in it?
- Over what time span is the average calculated? Does it move?
- How are the numbers rounded? How is the tie broken?

Remember, the most hidden assumptions are those you yourself hold. As a tester, you need to challenge yourself and question everything.

the limits of automation

Automation testing is frequently evangelised as the cure-all of software quality woes. However automated testing has limits on its effectiveness. Understanding these limits will keep us from trying to automate something that should never be.

Scoping Limitations

In an automated test, deviations from the norm are not necessarily reported as failures. We can work around that by writing more tests, each of them focusing on one factor of the system.

Practical Limitations: automation comes with a maintenance cost as the product evolves. This places practicality limits around what we automate. It's not feasible to automate everything, as we must maintain everything. We need to be prudent about what tests we want to keep.

Technological Limitations: some testing activities are just not possible to automate, like user experience testing. As soon as we move into the area where subjective qualities are being measured automation breaks down.

Usefulness Limitations: automated tests do not provide equal value to the team. The high use post-deploy smoke test is more valuable than checking whether the user name field supports "Travis" as well as "Cornelius". Just like we risk and value assess our manual testing effort we should be doing the same with automation.

Conflicting Limitations

The limitations we touched on they fall into two categories; those that force us to be smarter about the small set of tests we automate and those that drive us to want more tests. We can't have both. How we deal with this is what makes a good tester.