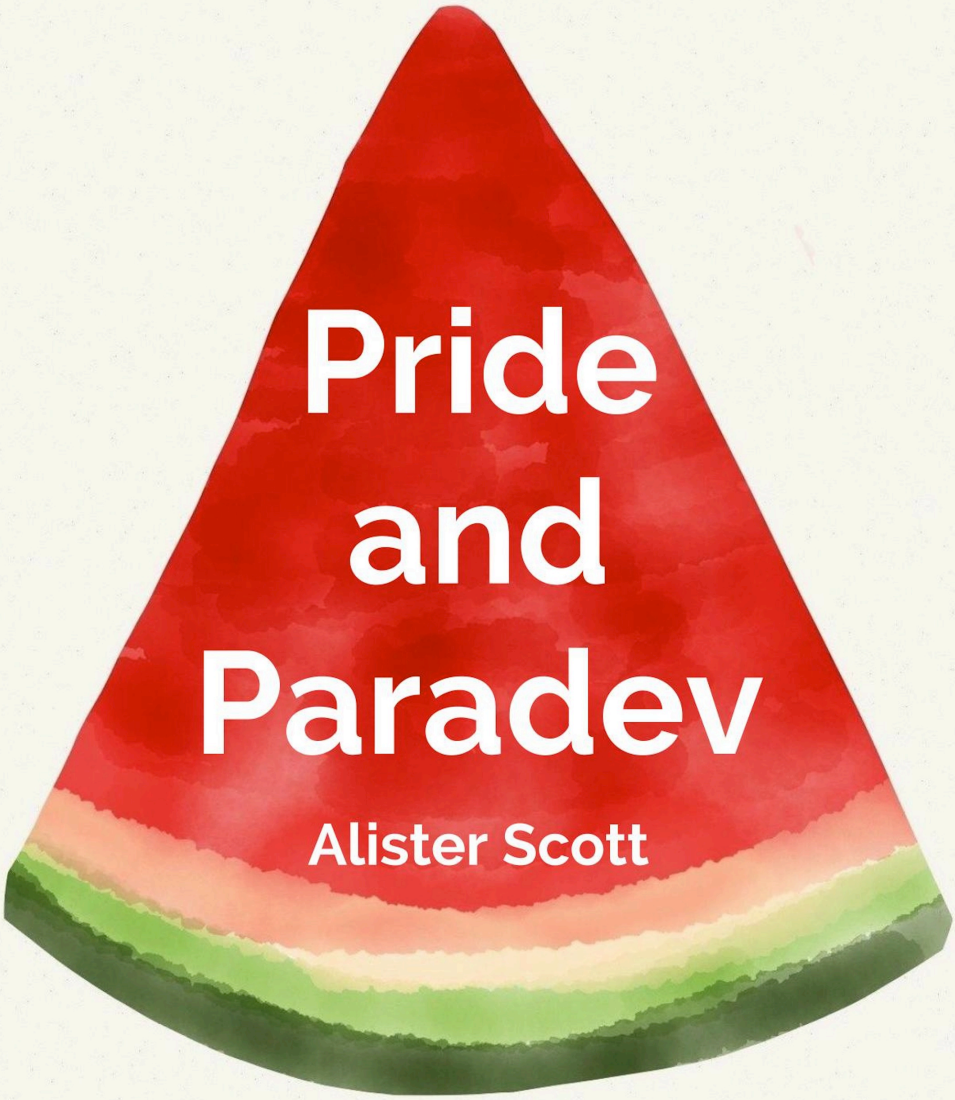


A collection of agile software testing contradictions

Pride and Paradev

a collection of agile software testing contradictions

Alister Scott

This book is for sale at http://leanpub.com/pride-and-paradev

This version was published on 2016-04-13

Contents

Acknowledgments

This book began as a series of blog posts on my WatirMelon[1] blog.

This was great, as I was able to get feedback on each article as I wrote it. I would like to thank my blog readers for the plentiful feedback I received; it has no doubt shaped the final release of this book.

I would also like to thank my wonderful wife Clare who reviewed this book and, despite having no prior knowledge of the subject matter, was able to provide a lot of excellent feedback and polish.

Finally, I'd like to thank[2] my family: Clare, Finley, Orson and Winston: you guys mean the world to me.

[1] http://watirmelon.com

[2] or retroactively apologize for spending all my time writing this, but at least I wrote your names at the start of my book!

About this book

Who is it for?

This book for anyone who *does* or *wants to do* software testing on an agile[3] team.

Full of Quotes

> "One must never miss an opportunity of quoting things by others which are always more interesting than those one thinks up oneself."
>
> ~ Marcel Proust

> "Quotations when engraved upon the memory give you good thoughts."
>
> ~ Winston Churchill

Why contradictions?

> "The test of a first rate intelligence is the ability to hold two opposed ideas in the mind at the same time, and still retain the ability to function."
>
> ~ F. Scott Fitzgerald, *The Crack Up*[4]

[3] http://en.wikipedia.org/wiki/Agile_software_development
[4] http://www.esquire.com/features/the-crack-up

Most software testing books are a verbose collection of *best practices*: what you should do as tester to be successful, often in a traditional 'lessons learned' format.

Best practices are often sold, particularly by consultants, as *silver bullets*[5]. I have particular disdain for best practices, they're not contextual and too black and white for me. I tend to see the world in shades of gray, a best practice in one context makes no sense in another.

> "The color of truth is gray."
>
> ~ André Gide

Writing books in shades of gray isn't easy: it's much easier to write a best practices book with strongly held views about what you consider to be the right approach, and dispel ideas that contradict your own.

In thinking of books in the middle of two views, one approach is to write a book sufficiently nebulous or generalist that it could appear to work in various contexts. But the outcome is weak as people prefer reading stronger views.

Instead I've decided to write a collection of contradictory claims about software testing; the practical implications lie somewhere in between.

I see this book as a bit of an experiment: I've certainly never seen a book following this format before, but who knows, it may create a whole collection of books with contradictions about software development.

[5] A silver bullet is a metaphor for any solution of extreme effectiveness.

The biggest benefit I have found in this approach is that it's particularly difficult for a reader to argue with me about something, because I am essentially arguing with myself!

So, what lies ahead is a book of what George Orwell dubbed *doublethink*, the act of simultaneously accepting two mutually contradictory beliefs as correct.

> "To know and not to know, to be conscious of complete truthfulness while telling carefully constructed lies, to hold simultaneously two opinions which canceled out, knowing them to be contradictory and believing in both of them, to use logic against logic, to repudiate morality while laying claim to it, to believe that democracy was impossible… to apply the same process to the process itself – that was the ultimate subtlety; consciously to induce unconsciousness, and then, once again, to become unconscious of the act of hypnosis you had just performed. Even to understand the word 'doublethink' involved the use of doublethink."
>
> ~ George Orwell, *1984*

Enjoy.

Before we start: what's a paradev?

What do you think of when you hear the word *paradev*?

I'll try to explain *my* meaning of the term with a true story.

A couple of years ago now, just after I started at ThoughtWorks[6], I read a tweet from a fellow programmer here in Brisbane along the lines of "the paradevs at work enjoyed my lunchtime session on networking". My ears pricked: "what's a paradev?" I asked. "It's someone who helps the developers develop" she replied. "Oh" I thought.

I admit my initial reaction was shock, what does that term even mean? The first things that came to mind when thinking 'para' were parasites, paraplegics, paralysis, paranoia, but these weren't related at all to software development. She told me she came up with the term to describe non-programmers who work in software from professions such as *paramedics* and *paralegals*, basically jobs that function to support a higher paid, higher level profession such as doctors and lawyers. I was offended, how dare she call me that.

But that was then and this is now. I've since reappropriated[7] the term, much like the term Queer[8] was reappropriated two decades

[6] http://thoughtworks.com

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

[8] http://en.wikipedia.org/wiki/Queer

ago. I'm happy to be seen as someone who helps the devs: "if I'm going to be a paradev: I'm going to be the best darn paradev there is!"

The story gets even better though. Very recently I was telling this story to a fellow paradev who, like everyone else I tell the story to, hadn't heard of the term. I saw him embark on some quick etymology to discover that para[9] is also used to indicate "beyond, past, by" (think *paradox*: which translates to *beyond belief*). This same reasoning translates paradev into *beyond dev* or *past dev*. How apt!

He joked with me that paradevs are the people on the team that don't box themselves into a narrow definition, happy to be flexible, and actually are happy to work on different things. Amen to that.

So to answer my original question: a paradev is anyone on a software team that doesn't just do programming. This book is for paradevs who do, or would like to do, software testing on an agile team.

[9] http://dictionary.reference.com/browse/para-

All about agile software development

This book is all about agile software development[10]. I will briefly discuss what agile software development is and why it's important (and fun).

Agile software development is all about delivering business value sooner

That's why we work in short iterations, seek regular business feedback, are accountable for our work and change course before it's too hard.

Agile software development is all about breaking things down

> "The secret of getting ahead is getting started. The secret of getting started is breaking your complex overwhelming tasks into small manageable tasks, and then starting on the first one."
>
> ~ Mark Twain

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

- A problem is too big so let's break it down into a project
- A project is too big so let's break it down into iterations
- An iteration is too big so let's break it down into user stories
- A user story is too big so let's break it down into acceptance criteria
- An acceptance criterion is too big so let's break it down into some tests
- Let's write a failing test and make it pass

Agile software development is all about automated testing

Delivering software everyday is easy. Delivering *working* software everyday is hard.

The only way an agile team can deliver working software daily is to have a solid suite of automated tests that tells us it's still working.

The only way to have reliable, up-to-date automated tests is to develop them alongside your software application and run them against every build.

Agile software development is all about communication and flexibility

You must be extremely flexible to work well on an agile team. You can't be hung up about your role's title. Constantly delivering business value means doing what is needed, and a team of

people with diverse skills thrives as they constantly adapt to get things done. Most importantly flexibility means variety which is fun!

A typical agile software development process

Some context

This is *not* a book about how to do agile software development.

The purpose of this chapter is to provide some information on the basics of agile software development using a fictional example project. Most of the terminology I use originates from the Extreme Programming (XP)[11] methodology. If you are after more in depth information on agile software development I recommend reading Jonathan Rasmusson's excellent book *The Agile Samurai*[12].

Beautiful Tea

Janet and Dave own *Beautiful Tea*: a boutique tea estate in the Byron Bay hinterland in Australia. They grow and sell organic loose leaf tea direct to tea connoisseurs by email and telephone.

Janet and Dave can no longer sustainably grow their business using email and telephone alone and have recently initiated a project to establish a bespoke online sales plat-

[11] http://en.wikipedia.org/wiki/Extreme_programming

[12] http://pragprog.com/book/jtrap/the-agile-samurai

form.

Project Inception

A typical agile software project often begins with a project inception workshop where stakeholders get together for a couple of days to plan the project and make sure everyone is in agreement with scope and priority.

The output of a project inception is a master story list, which is essentially a *prioritized* to-do list for your project broken into high level features and further broken down into user stories which have estimated effort against each.

Beautiful Tea Project Inception

Beautiful Tea recently held a two day project inception workshop where they spent time with their recently hired technical staff to incept the project by developing, among other materials, a Master Story List of features and user stories for their online ordering application. Each of the high level features was decomposed into user stories each with estimated effort in days (d).

1. Domestic Sales (20d)
    - List Products (5d)
    - Manage Inventory (4d)
    - Select Products (4d)

    - Shipping Selection (2d)
    - Credit Card Payment (3d)
    - Paypal Payment (1d)
    - Tracking (1d)
2. International Sales (7d)
    - Product Currencies (4d)
    - Shipping Selection (2d)
    - Tracking (1d)
3. Account Mgmt (13d)
    - Create (5d)
    - Edit (3d)
    - Delete (3d)
    - Newsletter (2d)
4. Reseller Ordering (10d)
    - Discount rates (5d)
    - Invoicing (3d)
    - Returns (2d)

Agile Iterations

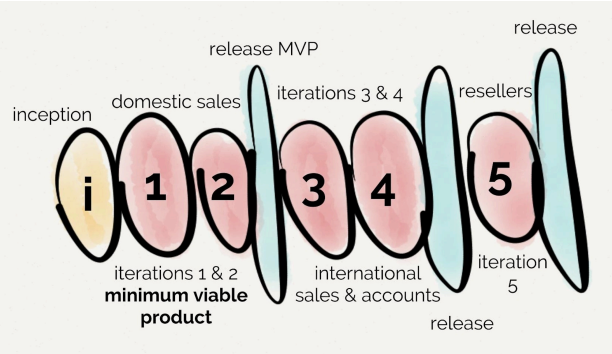
The Master Story List produced in an inception can be used to plan an agile project. An agile project is typically broken into iterations, which are typically 1-2 week blocks of time to deliver a number of user stories to the business. A *showcase* is delivered at the end of each iteration that showcases working, fully tested software to the business. The *velocity* of the project is the number of user stories delivered per iteration.

As the user stories are prioritized, the team can identify a milestone which marks the bare necessary functionality to release

to production - this has become known as the minimum viable product or MVP[13].

Beautiful Tea Project Iterations

Beautiful Tea split their agile project into five two-week iterations (10 days per iteration), with a Production release planned after the second, fourth and fifth iterations. It was determined by the team that the Domestic Sales feature was the minimum viable product as International Sales, Account Management and Reseller Ordering could be released later to enhance the online ordering experience and would impede the release of Domestic Sales.



[13] http://en.wikipedia.org/wiki/Minimum_viable_product

A user story

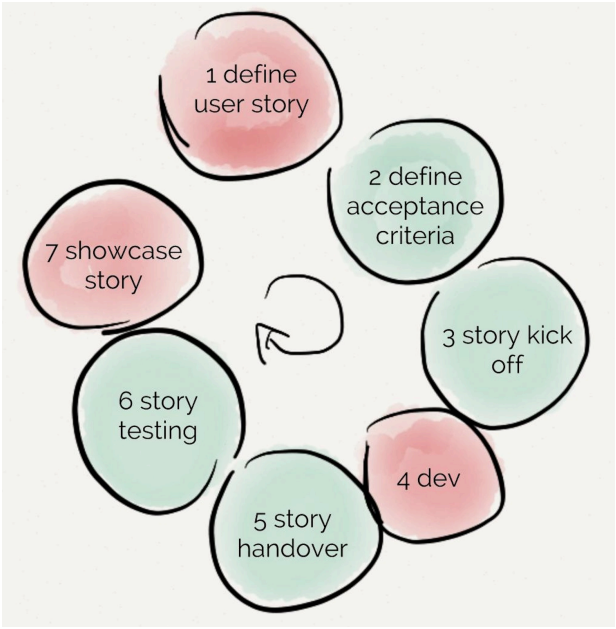
A user story consists of a narrative (background) and acceptance criteria (plus supporting material such as screen designs or necessary database information).

Typically a business analyst works with the product owner to define the acceptance criteria. If there is no business analyst on the team, the tester often takes on this role.

When a programmer or pair of programmers start work on a story, often they will have a quick story kick-off to discuss the acceptance criteria to make sure they are clearly defined and able to be developed.

The programmers write automated unit and integration tests alongside delivering the production code, and once complete, conduct a story handover at their workstation to demonstrate all acceptance criteria have been met by what has been developed. Depending on the team, the programmers may also write the automated acceptance tests.

The tester then tests the story against the acceptance criteria and in different browsers/devices/operating systems as needed. The tester may also conduct some exploratory testing and review any automated tests to ensure there's sufficient business and code coverage. Once testing is complete, the story is moved to *done* and is ready to showcase to the business. Once the necessary number of stories have been finished and showcased, these can be released into Production.



The user story life cycle begins with definition of the user story and acceptance criteria which involves the business analyst and tester. Once these are complete, a pair of programmers begin the story with a kick off which is a ten minute discussion between the BA, tester and programmers to discuss the story and acceptance criteria to ensure everything is clear and correct.

Coding takes place using test driven development, meaning a failing unit test is written before each piece of functionality is developed. The programmers also are respon-

sible for developing automated integration and acceptance tests.

Once coding is complete, a story handover takes place between the programmers and the tester and optionally the BA. This happens at the programmers machine and is a quick ten minute demo of the functionality working on a local environment.

Story testing is conducted by the tester who tests all acceptance criteria and edge cases in an integrated environment against various browsers. The tester also conducts exploratory testing to ensure requirements weren't overlooked and existing functionality still functions appropriately.

Any bugs are written on pink sticky notes and discussed with the programmers who promptly fix them. The user story remains in test as this happens.

Once the tester is complete, the story is added to the done column on their story wall ready to be showcased to the business representatives.

A collection of software testing contradictions

"It was the best of times, it was the worst of times"

~ Charles Dickens: *A Tale of Two Cities*

Your role as an agile software tester

> “Do not worry about holding high position; worry rather about playing your proper role.”
>
> ∼ Confucius

Do agile teams even need a software tester?

I admit this topic is a little strange to have in a book about software testing. But I thought I would include it nonetheless as it's relevant to our industry and it's good for you to have some background information about the reasoning behind hiring testers.

Agile teams don't need a software tester

You've probably heard the story. Facebook, of the most popular sites on the whole Internet (as of writing) has no testers. The Facebook engineer responsible for the feature is responsible for the testing.

This is because Facebook, by and large, does not need to produce high quality software.[14] They ship quickly and as a result, they ship bugs. Sure, they've got a lot of automated tests, but we all know there is still a need for human testing.

Evan Priestley, an ex-Facebook engineer, explains[15] that Facebook get around having no testers by doing a few things:

[14] Whether this will always be the case is another matter.

[15] http://www.quora.com/Is-it-true-that-Facebook-has-no-testers#

- They rely on extensive dogfooding[16] by internal engineers;
- They have extensive real time production monitoring for faults;
- They release code to a beta site 24 hours before release where major clients are forced to do QA testing (to avoid integration problems); and
- They provide channels for ex-employees to report bugs.

What can we learn from this? If you don't particularly care about quality, have good production monitoring, and can get internal engineers and major partners to do your QA then you *may* be able to get away with not having a tester on your agile team.

Agile teams definitely need a software tester

Most agile teams and product companies sooner or later realize they need a software tester.

Software testers provide a unique questioning perspective which is critical to finding problems before go-live. Even with solid automated testing in place: nothing can replicate the human eye and human judgment.

I've noticed that a lot of organizations that typically didn't have any software testers have started to hire or dedicate staff as testers as they begun to see the benefits of testers, or have starting feeling the pain of their absence.

[16] A term which means a company who uses its own products to demonstrate the quality and capabilities of the product.

Take 37signals[17], the web development company founded in 1999, who only in 2013 created a dedicated QA role. This news was fairly well hidden in a 37signals blog post introducing a new support member:

> "You may have noticed a picture of Michael's new working environment[18] a few months ago. We recently integrated QA testing into our development process, with Michael taking the lead. His effort has prevented potential problems and bugs in every new feature in Basecamp. Look for more details about this in the future."
>
> ~ Joan Stewart, *37signals*

Another example of relatively new introduction of testing roles is the Wikimedia Foundation who only recently hired a tester (or three) into their organization to take the lead on testing Wikimedia products including Wikipedia.

If your team is too small to support a full time dedicated tester, then look for somebody who can do both software testing and business analysis. That way you still have somebody who can be responsible for advocating quality, but you don't have to grow your team size unnecessarily.

[17] http://37signals.com

[18] http://37signals.com/svn/posts/3359

Do agile software testers need technical skills?

By technical skills I mean things like the ability to:

- read, understand and construct XML;
- read and write SQL scripts to query a database;
- understand the DOM of a browser;
- understand and construct regular expressions;
- understand basic coding principles and structures; and
- use version control systems.

Software Testers Need Technical Skills

> "Man is a tool-using animal. Without tools he is nothing, with tools he is all."
>
> ~ Thomas Carlyle, *Sartor Resartus*

You're testing software day in and day out, so it makes sense to have an idea about the internals of how that software works.

That requires a deep technical understanding of the application. The better your understanding of the application is, the better the bugs you raise will be. If you can understand what a stack trace is and why it's happening, the more effective you'll be in communicating what has happened and why, which makes fixing it easier.

> "Most good testers have some measure of technical skill such as system administration, databases, networks, etc. that lends itself to gray box testing."
>
> ~ Elizabeth Hendrickson, *Do Testers Have to Write Code?*[19]

As you're testing, you can easily dive into the database and run some SQL queries to make sure things actually did what they were meant to, or discover and test an exposed web-service using different combinations as it'll be quicker than testing the user interface and provides the same results.

You'll know IE7 JavaScript quirks and will be able to communicate these to a programmer and work on a solution that gracefully degrades[20].

Gone are the days where you'd be emailed a link to a test environment somewhere that you'll use to conduct some manual testing and provide some feedback. More often than not, you'll start by setting up your own integrated development environment on your own machine so that you can get all the latest

[19] http://testobsessed.com/2010/10/testers-code/

[20] "Graceful degradation means that your Web site continues to operate even when viewed with less-than-optimal software in which advanced effects don't work" ~ Peter-Paul Koch, Fluid Thinking

changes as they're committed by programmers and find issues sooner than during a testing phase.

You'll also probably be asked to build a test environment that other people can use, and a continuous deployment pipeline to automatically update that environment when appropriate.

Without technical skills you're going to struggle with this, as it's not just a matter of testing' the *functionality* of the application, but testing the entire *system*: that it can be built, deployed, internationalized, scaled etc.

Soon you'll start coming across other testing challenges such as how to test internationalization and localization[21], accessibility[22] and how to locate or generate appropriate test data. This may involve writing your own SQL scripts that take field labels and translate them to a test locale[23] to check screens for hard coded data. Again, these activities require technical skills.

Often programmers will show disdain for testers without any technical skills as they won't understand the technical challenges a programmer faces, and the timeframes needed to deal with complex issues. Testers without technical skills may struggle communicating issues in a technical way to programmers.

The more technical skills you have in your toolbelt, the more effective you can be as a software tester.

But having strong technical skills and wanting to do nothing but programming as the sole tester on a small agile team is a recipe

[21] http://watirmelon.com/2013/05/30/internationalization-and-localization-testing/

[22] http://watirmelon.com/2013/02/12/automated-local-accessibility-testing-using-wave-and-webdriver/

[23] a test locale is a specific language for testing that the correct elements of your application have been internationalized appropriately.

for disaster.

Software Testers Don't Need Technical Skills

> "A particularly terrible idea is to offer testing jobs to the programmers who apply for jobs at your company and aren't good enough to be programmers. Testers don't have to be programmers, but if you spend long enough acting like a tester is just an incompetent programmer, eventually you're building a team of incompetent programmers, not a team of competent testers."
>
> ~ Joel on Software on Testers[24]

Hiring testers with technical skills over having a testing mindset is a common mistake. A tester who primarily spends his/her time writing automated tests will spend more time getting his/her own code working instead of testing the functionality that your customers will use.

In a small agile team of say seven programmers and one tester, the tester will spend nearly all his/her time conducting exploratory and story testing so there will be no time to spend as a tester writing automated tests, it will need to be done by the programmers as part of developing a story. Hiring a tester who expects to predominantly write code on a small agile team is a big mistake.

[24] http://www.joelonsoftware.com/items/2010/01/26.html

> "Since testing can be taught on the job, but general intelligence can't, you really need very smart people as testers, even if they don't have relevant experience."
>
> ~ Joel on Software on Testers[25]

What technical skills a tester lacks can be made up for with intelligence and curiosity. Even if a tester has no deep underlying knowledge of a system, they can still be very effective at finding bugs through skilled exploratory and story testing. Often non technical testers have better shoshin[26]: a lack of preconceptions, when testing a system. A technical tester may take technical limitations into consideration but a non technical can be better at questioning why things are they way they are and rejecting technical complacency[27].

Often non-technical testers will have a better understanding of the subject matter and be able to communicate with business representatives more effectively about issues.

You can be very effective as a non-technical tester, but it's harder work and you'll need to develop strong collaboration skills with the development team to provide support and guidance for more technical tasks such as automated testing and test data discovery or creation.

[25] http://www.joelonsoftware.com/items/2010/01/26.html

[26] http://en.wikipedia.org/wiki/Shoshin

[27] This is really what I consider the core role of a tester to be: always questioning whether the system or functionality does what it actually should do.

Are software testers the gatekeepers or guardians of quality?

Software Testers are the Guardians and Gatekeepers of Quality

> "Quality is value to some person."
>
> ~ Jerry Weinberg

Working as a sole tester in a small agile team, you are the guardian of quality. You care about quality and it's your job to fight the right fight to ensure it prevails. As Jerry Weinberg said 'quality is value to some person' and you need to ensure that value is realized.

User stories aren't *done* until you've tested each of them, which means you get to provide information to the business about each of them. You define the quality bar and you work closely with your team and the business representative(s) to strive for it.

You'll soon realize that's it's better to start *building quality in* rather than *testing it in*, so make sure there are clearly defined acceptance criteria which have been marked as completed so that testing is more focused: efficient and effective. You'll work with the programmers to make sure that as many acceptance tests

are automated along side the code so that the regression testing burden is lessened each time a story is delivered.

Whilst the business ultimately wants a great product, you're working with the programmers closely in your team to ensure this happens on a day to day basis. You are the guardian of quality and they'll ultimately respect you for making them look good.

Software Testers aren't the Guardians and Gatekeepers of Quality

Whilst you think you may determine the quality of the system, it's actually the development team as a whole that does that. Programmers are the ones who write the good/poor quality code.

Whilst you can provide information and suggestions about problems: the business can and should overrule you: it's their product for their business that you're building: you can't always get what you consider to be important as business decisions often trump technical ones.

You're not perfect. Everyone is under pressure to deliver and if you act like an unreasonable gatekeeper of quality, you'll quickly gain enemies, whether that be business representatives or programmers, and people will simply go around or above you.

You can still be an advocate of quality in your team without being a gatekeeper: you just need to do it in a gentler way.

Should agile testers fix the bugs they find?

Software testers should fix the bugs they find

I admit it, I often fix bugs I find. I can't help it.

After being on a project for a couple of months you start to notice the same trivial bugs being found again and again. If you know how to fix it, why not fix it?

An example is a trailing comma in a JavaScript array[28], it'll go berserk in IE7, and it's easy enough to fix (remove the trailing comma) so I'll just fix it if I find it. Another is an button in IE7 that is meant to submit a form but won't[29]. To support IE7 you need an input type=submit to work. Again, I'll change it so that it works.

The benefits are that user stories will move to 'done' faster as it doesn't require programmer involvement, and it's less disruptive to the programmer who will be working on another story and will need to context-switch.

[28] http://stackoverflow.com/questions/7246618/trailing-commas-in-javascript
[29] http://stackoverflow.com/questions/4020187/ie7-button-does-not-submit-form

Software testers should not fix the bugs they find

You'll often be tempted to do a quick bug fix when you know why something is broken, but you should avoid it. If you quickly fix it, the programmer who created the bug doesn't get the feedback that they made a mistake, and will repeat the same mistake over again.

Over time, if the programmers know that you'll fix bugs, they'll naturally start providing you with buggier code as they know that you'll just fix it as needed.

Programmers crave feedback, both positive and negative, that's why it's good having a tester on an agile team. But fixing bugs yourself means there's less feedback being given, and less communication happening.

There's also a small chance that you may introduce some regression bugs when fixing a bug by yourself, but this can be countered by adding an automated regression test.

Another minor reason is that it may look like you're not finding bugs, but this again shouldn't be reason alone because testers shouldn't be measured on how many bugs they find.

Should testers write the acceptance criteria?

Testers should write the acceptance criteria

The main benefit of having a tester write acceptance criteria is that they are more likely to be measurable and testable as whilst writing them the tester will be asking themselves: how will I be able to test this?

It also encourages a good working relationship between business representatives and testers, and programmers and testers, which can come in very handy when conducting testing if there are issues with implementation, as they will have the background knowledge of what was originally required.

Testers will also be able to think of unusual edge cases in the acceptance criteria which can be considered from the start.

There may not be a business analyst on your team and in this case it makes sense for a tester to take on the responsibility for writing the acceptance criteria, so that the programmers can focus on implementing them.

Testers shouldn't write the acceptance criteria

Testers like to think of how things shouldn't work more than how they should work. Sometimes testers will be more pedantic about the acceptance criteria and can get carried away with them – adding niche edge cases which might not be applicable to the real world, or just not a priority to the business, who is trying to get the thing out the door and delivering business value.

More often than not a bandwidth limitation will stop a tester from writing acceptance for user stories. Working as a solo tester on an agile team, having to both write the acceptance criteria and test the acceptance criteria, often in multiple browsers and/or devices, can be a time consuming thing, so in this case it would be advisable to get someone else to focus on writing the acceptance criteria so that they can be properly tested.

Software testing as a career choice

> “A career is wonderful, but you can’t curl up with it on a cold night.”
>
> ∼ Marilyn Monroe

Is software testing a good career choice?

Software Testing is the Worst Career on the Planet

It's amazing how quickly you tire of testing the same thing over again in Internet Explorer 7 because the programmers don't use Internet Explorer and hadn't thought to test it in that.

The harder you work at finding bugs the lazier the developers become at letting them through.

People constantly question you about why you're *still* a software tester and haven't turned into a programmer yet as though technical specialism is a natural career progression.

Lots of people call themselves software testers because they've *played* with software over a couple of years and attended a testing certification course over a couple of days. You're grouped into the same category as those people.

Just when you think you've got a user story tested in three different operating systems, four devices and eight browsers, the programmer decides to 'refactor' their code, or switch to a more in vogue JavaScript framework, rendering all your testing work void because every screen you have tested no longer functions.

And they expect you to test it by the end of the iteration which happens to be today.

Despite what iterative development brings testing always gets squeezed and you're expected to constantly go above and beyond to get things done, in minimal time, with an emphasis on high quality and you'll often ultimately get the blame when things go wrong.

Career progression means either becoming a specialist 'automated tester' or a test manager, one involves writing code, that no one ever sees, the other usually involves writing wordy template driven *test strategies*, again, that no one ever sees.

But the absolutely worst thing about being a software tester is the distrust you develop in software. You constantly see software at its worst: it's hard to believe that any software can be developed that actually works without any issues. This means you hold a deep breath every time you hit *submit* on a credit card form, praying that it will actually work and not crash and charge your credit card three times.

Software Testing is the Best Career on the Planet

Some days I am amazed at how much fun my job is. I get to play with cool gadgets: I have four smart phones and an iPad on my desk, use three operating systems and eight browsers on a daily basis.

I get to look at software from all different angles: from a user's point of view, from the business/marketing view, from a technical viewpoint and try all kinds of crazy things on it.

I get to really know and understand how a system works from end-to-end, and get to know its quirks and pitfalls. Finding bugs prevents them from being released into Production and causing someone else a great inconvenience.

I develop great relationships with programmers who value the feedback I give, and business people who I work with to develop acceptance criteria and discuss issues in business terms and how they will be effected.

I get to understand code, database schema, servers and browsers. I am involved in automating acceptance tests. I get to go to awesome software testing conferences around the world[30] to meet other testers.

I get to tell[31] my family about all the cool things I've tested and they get excited to occasionally see things I have worked on in the media etc.

It's a really cool career.

[30] http://watirmelon.com/2013/02/18/going-to-gtac-in-nyc/

[31] and show them the cool stuff I have worked on when it is a public facing application.

Is it beneficial to attend software testing conferences?

You should attend software testing conferences

Software testing conferences are a fantastic opportunity to meet other testers in person and discuss challenges you face and come up with ideas on how to do things better. You may get to know other testers online and if you get to meet them in person at a conference it will strengthen your working relationship.

Your employer may have a training budget for each staff member, and if you're a self-motivated learner like me, instead of using your training budget to attend actual training, or a testing certification, you can instead use it to attend a software testing conference where you'll learn a great deal more real world knowledge. I use my training budget each year to attend one overseas conference, so I am selective in which one I attend.

One of the biggest benefits of attending a software testing conference is realizing that you and your organization are not unique in the testing problems that you face by hearing from others in similar situations, and hearing of suggestions to overcome them.

You don't need to attend software testing conferences

Conferences are expensive to attend as often they're in a different city/country to you and you have to pay for travel and hotel costs, as well as tickets. It will often mean time off work also, which can be a challenge to arrange as a contractor, if you're casually employed or a consultant working on a client site with a deadline.

Lots of conferences now stream their talks live for free, and most speakers publish slides or videos afterwards, so you don't need to specifically attend to capture the knowledge the speaker presents, but dedicating some time to do this is critical, otherwise it'll just remain on your to-do list.

A lot of software testing conferences fill slots with 'serial speakers' who speak at every conference they can, often with recycled material that you may have seen in some form or another at some point elsewhere. If you read the blog of one of these speakers, chances are the content of their talk will already be available in a slightly different form.

The worst part of attending a software testing conference is having to put up with potential bad behavior of other attendees. As we all know, people at conferences can be in bad form, which has led to many conferences now having a explicit code of conduct[32] for attendees, which makes the sensible people feel like children.

Many conferences are also drinking parties in disguise, so if

[32] http://confcodeofconduct.com/

you're a teetotaler like me then you'll often feel out of place, and bored, as soon as the partying/drinking begins.

Should testers get a testing certification?

There are numerous software testing certifications available to certify the skills of software testers. These typically involve some training followed by a multiple choice examination.

Some examples are ISTQB[33] and ISEB[34]

By testing certifications I mean any of these.

So, should testers get a testing certification?

Get a testing certification

Get a testing certification if you feel like getting a testing certification will be useful.

Don't get a testing certification

Don't get a testing certification if you don't feel like you need a testing certification.

[33] http://www.istqb.org/

[34] http://certifications.bcs.org/

Defining Acceptance Criteria for user stories

> “Happiness can exist only in acceptance.”
>
> ~ George Orwell

Should acceptance criteria be implicit or explicit?

> "Understanding is the knowing of misunderstanding"
>
> ~ Zivarnna Smithies

Acceptance criteria should be implicit

All things in life are implicit. When my wife asks 'can you go to the shop and get me some milk', she doesn't also have to tell me 'and don't buy anything else in the store', 'and make sure it's 2 liters of 2% fat pasteurized non-homogenized cow milk', 'make sure it's refrigerated' and 'pay with cash and use our Flybuys card'. These are *implied* criteria from our shared understanding.

Acceptance criteria are the same. If I had to explicitly state every acceptance criterion for every story; writing acceptance criteria would be a never ending task.

Recently I have, reluctantly, started including acceptance criteria that really should be implied: 'page has page title', 'page passes WAVE[35] accessibility' etc. But where do you draw the line? Next I'll be writing "works in a web browser", and "works on the Internet".

[35] http://wave.webaim.org

Keep acceptance criteria focused on what is required, not what is obvious.

Acceptance criteria should be explicit

> "It pays to be obvious, especially if you have a reputation for subtlety."
>
> ~ Isaac Asimov

> "The obvious is that which is never seen until someone expresses it simply."
>
> ~ Kahlil Gibran

If you don't have explicit acceptance criteria then it can be very hard to determine whether what has been developed is actually completed.

The more effort you put into writing clear, explicit acceptance criteria is well rewarded by receiving a story that has all required functionality included in it. This means less rework because each time you find an acceptance criterion induced bug it means the programmer has to context-switch to fix it which is less time spent working on a new story.

Conducting a user story initiation with the programmer before development, and a developer-tester dev-box walk-through as soon as development is finished is a great way to ensure that the acceptance criteria you have written is explicit, and has been implemented exactly as intended.

Even though some things are obvious requirements (eg 'page passes WAVE[36] accessibility'), these are often forgotten so it pays to have a standard list of common acceptance criteria and include them on every user story.

[36] http://wave.webaim.org

Should your acceptance criteria be specified as Given/When/Then or checklists?

You should specify your acceptance criteria as Given/When/Then

Given/When/Then is almost a ubiquitous[37] way to specify user scenarios:

```
Given some precondition
When I do some action
Then I expect some result
```

If you write your acceptance criteria in this format, it not only provides a consistent structure, but if your automated acceptance tests are also specified in the Given/When/Then format then it makes translation from acceptance criteria to acceptance tests very efficient.

[37]This format was first written about by Dan North in 2007, and is sometimes (incorrectly) referred to as the Gherkin language, which is specific to the Cucumber software tool.

Your acceptance criteria are less likely to be nebulous as thought has gone into making them follow this consistent format with a precondition, action and expectation.

You should specify your acceptance criteria as checklists

Having acceptance criteria specified against a user story in checklist format makes the acceptance criteria clear and concise as there is nothing more needed. Given/When/Then scenarios can be verbose and hard to read on a user story card.

It also means these are easy to individually mark as complete by programmers as they implement the functionality (in a tool such as Trello[38]).

But the biggest benefit is that acceptance criteria checklists can't be transferred directly from the user story to an automated acceptance test. This is important because acceptance test features shouldn't replicate user stories: a user story is a change to a system, a feature is a collection of things that it does. So having a feature for every user story is a disaster, which is more likely to happen if your acceptance criteria are in the Given/When/Then format.

Having your acceptance criteria in checklist form also means there's some human thought about how to implement an acceptance test for these. It might be that one acceptance test scenario covers three acceptance criteria checklist items, which is how it should be done, not simply a one to one mapping

[38] http://trello.com

which can happen when using Given/When/Then format for both acceptance criteria and tests.

Are physical or virtual story walls better?

Physical story walls are better than virtual story walls

There's nothing like seeing the status of an iteration using a large story wall: several columns and colored index cards that move across as the iteration progresses. Avatars are stuck against cards as people work on them (which can limit WIP[39]) and it's very easy to see from a glance any bottlenecks as the cards literally pile up.

The act of moving a physical index card (representing a story) into the 'done' column is refreshing, as is ripping up a card when it decided that it will provide no value.

You can stick colored sticky notes to each user story in test to represent bugs which are *squashed* as they are fixed.

Daily stand ups are held around the story wall, and the physical story wall can be 'walked' during stand up to ensure that everything is up to date and nothing will be missed.

[39] http://en.wikipedia.org/wiki/Work_in_process

Online story walls are better than physical story walls

Physical story walls aren't good for capturing what is actually required to deliver user stories (writing notes on the back can be very nebulous and hidden). Online story walls (such as Trello[40] or Mingle[41]) are great at both displaying the status of an iteration, as well as storing details and artifacts about each story. These include acceptance criteria that can be marked off when done, as well as a list of bugs against each user story which are marked off when fixed. Prototype designs can be attached and referred to by the programmer/designer/tester who is working on the user story.

If you have remote team members you can't solely rely on a physical story wall, and even if you don't, if team members choose to work from home (for example – late at night) then an online story wall makes a lot more sense as it is accessible from anywhere you have an internet connection.

A large screen 27″ iMac makes a great machine to have located centrally in your team to display the always on online story wall for all to see and update, plus you could hold your daily stand up around it.

[40] http://trello.com

[41] http://www.thoughtworks-studios.com/mingle-agile-project-management

Testing Techniques

"No one tests the depth of a river with both feet"

~ African Proverb

Which is better: manual or automated testing?

Manual Testing is better than Automated Testing

Manual testing is better than automated testing. Even when automating a test scenario, you have to manually test it at least once anyway to automate it, so automated testing can't be done without manual testing. And you have to manually check the automated test results also.

Automated tests can be stopped from working by something as simple as an unexpected pop-up dialog which can be quickly analyzed and dismissed when manually testing.

Manual testing is a sapient[42] activity: one that requires human judgement. As you are testing you are using implicit knowledge to judge whether or not something is working as expected. This enables you to find extra bugs that automated tests would never find. It also allows you to follow smells you find to explore areas that may not have been tested or required.

Manual testing is also helpful for finding layout issues and trivial bugs which wouldn't be found by an automated test, as you're fully observing the application as you're using it. Usability issues

[42] http://en.wikipedia.org/wiki/Sapience#Sapience

are also identifiable by manual testing but can't be discovered through writing and running automated test scripts.

Automated Testing is better than Manual Testing

Automated testing is better than manual testing. Automated tests are very explicit (black and white) so you have a much higher chance of reproducing a bug if found by an automated test by knowing what the automated test executed to achieve the result. Because the automated tests are explicit, they also execute consistently as they don't get tired and/or lazy like us humans, as we're more prone to human error.

Automated tests are quicker to run than manual tests as there's no lag time between input and checking, and this means you can run more tests in more browsers more quickly. Manually testing the same functionality in, for example, 8 browsers and 4 devices is tiring, but can easily be achieved with automated tests.

Automated tests also allow you to test things that aren't manually possible. For example, answering a question like 'what if I had 200 accounts', or 'what if I processed ten transactions simultaneously' can only be answered efficiently by using automated tests.

Can we just test it in production?

With continuous deployment, it is common to release new software into production multiple times a day. A regression test suite, no matter how well designed, may still take over 10 minutes to run, which can lead to bottlenecks in releasing changes to production.

So, do you even need to test before going live? Why not just test changes in production?

Test changes in production

The website for The Guardian[43], the UK's third largest newspaper, deploys on average 11 times a day, of which all changes are tested in production.

> "Once the code is in production, QA can really start."
>
> "Sometimes deployments go wrong. We expect that; and we accept it, because people (and machines) go wrong. But the key to dealing with these kind of mistakes is not to lock down the process or extend the breadth, depth and length of regression tests.

[43] http://en.wikipedia.org/wiki/The_Guardian

> The solution is to enable people to fix their mistakes quickly, learn, and get back to creating value as soon as possible."
>
> ~ Andy Hume on *Real-time QA*[44] *at The Guardian Developer Blog*

The key to testing changes as soon as they hit production is to have real time, continuous real user experience monitoring[45]. This includes metrics like page views and page load time, which directly correlate to advertising revenue, an incentive to keep these healthy.

More comprehensive automated acceptance tests can be written in a non-destructive style that means they can be run in production. This means that these can be run immediately following a fresh production deployment, and as feedback about the tests is received, any issues can be remedied immediately into production and tested again.

This also means you don't need to own and manage test environments which can be costly and time-consuming, especially if you're constantly trying to keep them as *production* like as possible.

Test changes before production

There are a limited number of businesses that are able to release software without any form of testing into production: whether

[44] http://www.guardian.co.uk/info/developer-blog/2012/dec/06/real-time-qa-confident-code

[45] http://en.wikipedia.org/wiki/Real_user_monitoring

there be legislative requirements requiring testing, or the risk of introducing errors is too high for its target market.

Whilst automated regression tests do take longer to run than unit or integration tests, there are ways to manage these to ensure the quickest path into production. These strategies include running tests in parallel, only running business critical tests, only running against the single most popular browser, or only running tests that are directly related to your changes.

You can set up a deployment pipeline that runs a selected subset of tests before deploying into production then running the remaining tests (in a test environment). Any of the issues found in subsequent tests are judged to see whether they warrant another immediate release or whether they can be included in the next set of changes being deployed into production.

Whilst you definitely should run tests before deploying to production, it doesn't mean that this has to drastically hinder your ability to continuously deploy.

What type of test environment should we test in?

Use a local test environment

If you're working on automated tests as a tester then chances are you've got your application's code-base checked out and running locally on your machine.

It's easy to use this locally running code base to conduct your story testing. The benefits are that it'll be very up to date, and as a programmer commits a change, you can grab the changes and you'll be working on a fresh copy.

As part of your testing, you can change what you wish, like application settings and database configuration to see what happens as you test without having to worry about impacting anyone else using the test environment.

You can even fix some bugs as you find them (if you choose to).

This reduces the overheads of having multiple test environments for story testing, as test environments need to be configured and managed.

Use an integrated test environment

You should always test your user stories in an integrated test environment: that is, an environment that is centrally managed and integrated with other systems.

One reason is that when you're testing; you're not only testing your application, but that your application can be *deployed* and work in a dedicated environment. This means you'll find issues that don't appear locally – like forgetting to include a file to be deployed, which will not show when testing locally.

You also test that you can access your application over the network using an external IP address, rather than using localhost, and iron out any connectivity issues.

You're also testing how your application performs under more realistic hardware than on just your development machine.

It's relatively easy to set up an automated continuous delivery pipeline to deploy automatically to a test environment so you can test stories as they are completed by the programmers.

Should you use test controllers for testing?

A test controller is a way to directly access functionality in your web application without following the standard web application flow: for example, you may want to directly access the credit card details screen, so what you do is develop a 'credit card details controller' which sets the desired state (an order and a customer) and shows the credit card details screen to you. The test controller is hit via a test URL which sets the state and redirects to the appropriate page. These test controllers are either not deployed to production, or are disallowed via routing in production

You should use test controllers for testing

Test controllers make for very efficient and easy to read automated acceptance tests. Instead of something like:

```
Given I am on the home page
When I add some products to my basket
And I provide my customer details
And I select express shipping
Then I should see the customer details screen
When I submit an invalid credit card number
Then I should see an error
And the credit card details are empty and need to be \
reentered
```

This can simply become:

```
Given I am on the credit card details page
When I submit an invalid credit card number
Then I should see an error
And the credit card details are empty and need to be \
reentered
```

The Given step simply calls the test controller that sets up required state and provides the screen. The scenario is focused on testing one thing and isn't reliant on a process flow to set up the state.

Test Controllers also allow you to test something before other things are developed. For example, in the above case you could test the credit card details screen even though the shipping selection page was not developed. As long as the controller sets things up correctly, it allows you to test stories in isolation without having dependencies on other user stories, which can result in a faster team velocity.

You shouldn't use test controllers for testing

Test Controllers set up state in your web application, which may be different to the state set up by using the application itself. For example, a test controller for credit card details may set the shipping method, but it may set it differently to how the shipping method screen does it, which results in inconsistent behavior in your application, and potentially false positives. This leads to failures where you are not sure whether there is an bug in your application, or just in the test controller itself.

User's don't use test controllers so neither should you. It may be convenient to jump into testing a certain page but testing is as much about the journey as the destination, so jumping straight in may mean you miss important bugs along the way.

Should you use production data or generate test data for testing?

You should generate test data for testing

Generating test data is the only reliable way to accurately run tests repeatedly and consistently knowing that the input test data hasn't changed.

Some applications rely upon specific data which is either hard to find, or hard to fake. For example, the web application I am working on displays different promotions based upon which day of the week you are using the system, and also changes prices depending on the day of week and time of day.

If you were using production data for testing, you would either have to run tests at specific dates/times to test different promotions/prices, or you would have to change the server date/time to test these. Changing the date/time on the server will effect anyone else using that server, so should be avoided. It also means that as you run your automated tests continuously against new check-ins, if you don't use a known set of generated test data, you will get different results depending on time of day.

When developing an entirely new feature, there won't be production data that you can use for testing, so you will need to generate some in this case.

Generating specific test data will often take longer than sourcing production data, but will retrieve results over time as tests are run very consistently against a known data set.

You should use production data for testing

When you're testing a web application, you're as much testing the data as testing the application behavior. Using production data will ensure that what you are testing will be as close as possible to the actual behavior once the feature is released to production users.

If you generate test data and use it to test, who is to say that this test data is actually valid. If you generate test data through lower level means (such as SQL insert scripts), you may introduce test data that isn't representative of that in production that may either introduce errors in functionality when actually running against production data, or errors in test that won't actually exist in production. As your database schema updates and evolves, you will need to also keep your data generation scripts up to date so they are reflective of production at all times.

If you do use production data, you need to be clever about how to source data. Querying the database using SQL scripts is an effective approach as it will enable you to quickly find real data that you can use to verify a story has been implemented correctly.

It will also allow you to identify outliers and edge cases that can be tested using real production data against the system in development.

If there any privacy concerns about using production data for testing, these can be mitigated by obfuscating[46] the data so that it has no identifying features.

[46]this means to scramble the data which may include randomizing names, or changing dates of births and addresses etc.

Should you test in old versions of Internet Explorer?

Test everything in IE7

IE7 is a bug magnet: seriously, I find more bugs in IE7 than any other browser. Why? It's the least forgiving of browsers. If it works in IE7 it'll most likely work in a more modern browser. It's like a fussy relative: if they like a gift you give them, chances are your less fussy relatives will also like the same gift.

Even though only 2% of customers use IE7, in a high volume business such as Amazon or eBay this still equates to millions of dollars.

It's very easy to test in IE7. First you'll need Windows XP: set up a VirtualBox[47] Virtual Machine (VM) with Windows XP installed and immediately disable automatic Windows Updates (otherwise your IE7 machine will quickly become your IE8 machine).

I recommend using a real IE7 browser over a IE7 mode in IE9 or IE10. Why? IE7 mode in IE9 or IE10 is a *simulator* and doesn't behave exactly the same as a real IE7 browser.

[47] https://www.virtualbox.org/

I don't expect programmers to use Windows XP, so I just add a "works in IE7 mode" acceptance criteria to every story so that each programmer will test locally in IE7 mode on their development machine: this saves me time testing each story.

Don't test anything in IE7

Many organizations use current production browser statistics to determine which browsers to support and hence test against.

But you shouldn't base your browser on *current* usage: it should be based upon *expected* future usage, determined by studying browser usage trends. Using this method it's pretty clear that usage of IE7 is continuing to fall: so why bother testing in it at all?

Customers who use IE7 are probably the sort of customers who aren't going to spend a lot of money on your web application: otherwise they would've got a more modern computer by now.

Besides, using IE7 is a pain. You need a separate VM as you can only use it on Windows XP. It doesn't contain any developer tools and there's no JavaScript console: particularly handy for debugging nasty bugs.

Programmers will get annoyed with you when you raise an IE7 bug that doesn't repro in IE7 mode in IE9 or 10, as they can't debug it locally.

Is it worth testing anything in IE7? I say no.

Should you use a tool to track bugs?

Don't use a tool to track bugs

When working in a collaborative, co-located agile team working in a iterative manner, it's often more efficient to fix bugs as they're found than spend time raising them and tracking them using a bug tracking tool. The time spent to raise and manage a bug is often higher than actually fixing the bug, so in this case it's better avoided.

Most testers aren't comfortable with this approach, initially at least, because it may look like they're not raising bugs. **But a tester should never be measured on how many bugs they have raised**. Doing so encourages testers to game the system by raising insignificant bugs and splitting bugs which is a waste of everyone's time. And this further widens the tester vs programmer divide.

Once a tester realizes their job isn't to record bugs but instead deliver bug free stories: they will be a lot more comfortable not raising and tracking bugs. The only true measurement of the quality of testing performed is bugs missed, which aren't recorded anyway.

One approach I have taken is to simply record bugs on sticky notes or index cards stuck to the team's story wall. This is a

lightweight approach as the only time taken is to write the sticky note and once resolved, it can be scrunched into a ball: a symbolic act of squashing the bug.

Use a tool to track bugs

> "What's measured improves"
>
> ~ Peter Drucker

If you've got remote team members, you can't really avoid using a tool to track bugs. It ensures you're communicating and tracking the progress effectively across geographic borders.

Without some form of bug tracking tool in place on your project, it's difficult to keep a historical track of bugs and how they are resolved. Without this, it may lead to some of the nastiest bugs reappearing: I call these **cane toads**.

If you weren't aware, cane toads[48] are a highly invasive species of toad in Australia that were introduced to control native cane beetles, but have ended up threatening natural wildlife. They have two notable characteristics:

1. They secrete toxic poison affecting their surroundings; and
2. They have an uncanny ability to survive even the harshest of conditions (here in Queensland there are competitions to kill cane toads but they're amazingly hard to kill: just when you think you're done with one it'll bounce back to life).

[48] http://en.wikipedia.org/wiki/Cane_toads_in_Australia

Therefore, I see a cane toad on a software project as an issue that:

1. Causes other issues by secreting toxic poison; and/or
2. Seems to come back to life even though you're sure you've already killed and buried it before.

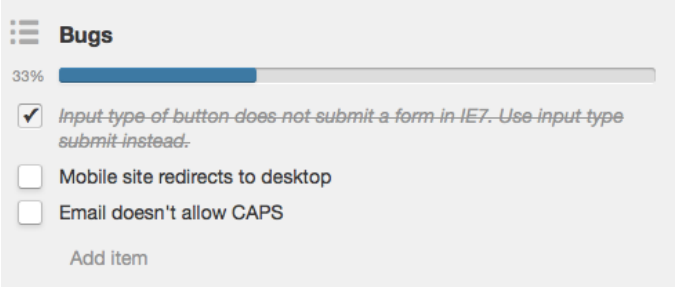
Without tracking these cane toads, and how you killed/fixed them, you'll see these reemerge. You can easily look up when it last happened, what you did then, and why it shouldn't be happening again.

This is also why it's really important to have automated regression tests.

You must keep bug tracking tools as lightweight as possible. These should allow rapid recording and resolution of bugs. Avoid the temptation to introduce 'defect workflow' of any description: empower people to do the right thing in regards to tracking and resolving bugs, don't make it harder than it should be.

An even better approach is to incorporate bug tracking into your existing user story tracking tool. Trello[49] allows checklists on user story cards: these are a great way to track bugs in a lightweight, flexible manner.

[49] http://trello.com



Trello Bug Tracking

Should you raise trivial bugs?

You should raise trivial bugs

Some of the world's best companies have become that way through attention to detail.

There are lots of famous stories about Steve Jobs when he was in charge of Apple about his pedantic nature. For example, how he would debate for half an hour about the shade of grey[50] for the bathroom signs in Apple Stores.

Another example is how he rang Google's Vic Gundotra[51] at home early on Sunday morning to let him know the shade of yellow of Google's logo was wrong on the iPhone but Steve had put an Apple engineer immediately on it.

All seemingly trivial things add together to create a broader perception of a company to consumers.

Raising trivial bugs is paying close attention to detail. Whilst customers might not notice something so little that is wrong; from little things big things grow.

[50] http://au.businessinsider.com/steve-jobs-attention-to-detail-2011-10?op=1#he-agonized-over-the-way-the-title-bars-at-the-top-of-files-and-windows-looked-1

[51] http://www.macrumors.com/2011/08/25/steve-jobs-called-googles-vic-gundotra-on-a-sunday-about-this-icon/

You should raise trivial bugs even if you choose not to fix them, that way you can keep a list of known bugs you will release into production should you choose to fix them one day.

You shouldn't raise trivial bugs

The problem with raising trivial bugs is that it slows your velocity and takes focus away from other things: namely building new functionality to release to customers.

> "You heard me right: An obsessive focus on quality can be a bad thing."
>
> ~ Jason Fried on *the importance of quick and dirty*[52]

If you focused your effort on fixing 100% of every issue ever identified with your application you will never actually ship the thing but continually try to perfect it. Fixing a trivial bug may then further highlight other trivial bugs and so the cycle begins.

By the time you release your application, a trivial bug may not even matter that much as it may be on a feature that won't even be used: but you can only find out this by releasing your product into production: trivial bugs and all.

[52] http://www.inc.com/magazine/201305/jason-fried/the-importance-of-quick-and-dirty.html

Should you involve real users in testing?

Yes, involve real users in testing

Unless you involve real users you risk releasing something into production that is not user friendly. User testing doesn't need to be expensive; you can conduct it in house with a focus on simplicity by reading Steve Krugg's excellent how-to guide *Rocket Surgery Made Easy*[53].

The caveat is that you need to conduct user testing as earlier as possible: there is no point doing user testing if it doesn't result in useful change, and the earlier you do it the more likely you are able to change.

You don't even need to have high-fidelity screens to conduct user testing: low fidelity screens are often enough to get critical usability and concept feedback from real users.

No, don't involve real users in testing

The problem with involving real users in testing is the risk that your organization will use this as an opportunity to listen to what users want.

[53] http://www.sensible.com/rsme.html

Listening to what users want is dangerous. This is because users think of things in *incremental* rather than *revolutionary* terms. Users don't know how to ask for something they've never conceived of. Listening to feedback from users is going to yield incremental improvements ('make that button green'), but this does in no way correlate to releasing something that users actually want.

In my pre-MP3 university days, I had a very large collection of audio CDs. I worked hard in a casual job and saved up and bought a Pioneer 25 CD stacker: it was amazing, I could listen to 25 CDs on shuffle! If I was asked what could have made listening to music better I would have said a 100 CD stacker: I could store four times as many CDs! A few years later Apple released a new thing called an iPod, a small pocket sized device capable of holding 1000 albums. If Apple had listened to users like me they would have built a faster, larger, better CD player. Instead they designed something I had no capability of thinking could even exist, I was thinking incrementally, Apple were thinking revolutionary.

You are better off understanding what users motivations are and then building something to satisfy that than involving them in user testing.

Automated Acceptance Testing

> "As machines become more and more efficient and perfect, so it will become clear that imperfection is the greatness of man."
>
> ~ Ernst Fischer

Do you need an automated acceptance testing framework?

Yes, you need an automated acceptance testing framework

If you're starting off with automated acceptance testing and you don't have some kind of framework, eg, page object models, in place then you can quickly develop a mess. If your automated acceptance tests are being written by various people on your project, then having a framework in place that people can follow will make for the most consistent approach.

There are certain operations that you can abstract to a base page class to ensure consistency across pages, and you can write helper methods for automated test drivers so that the same functionality is being repeated across your code base.

Without some kind of framework in place you're likely to have various approaches implemented which will eventually cause a maintenance overhead as your automated test suite expands.

No, you don't need an automated acceptance testing framework

There's an old saying in extreme programming: YAGNI[54]: you ain't gonna need it, which means a programmer shouldn't add functionality until absolutely necessary.

An automated acceptance testing framework violates this principle, there is a strong risk of developing functionality in your framework which you ain't gonna need.

Over-engineered automated acceptance test frameworks are harmful for a team as they dictate certain ways of doing things which means the team can be less efficient in developing what they need to deliver.

Developing a framework before any functionality is delivered is particularly inefficient, as it is not until you start using a framework you will understand what you require it to do and what it shouldn't do.

Pair programming on the automated acceptance tests can ensure a consistent approach is taken to development and knowledge across functional areas is shared.

[54] http://en.wikipedia.org/wiki/You_aren%27t_gonna_need_it

Who should write your automated acceptance tests?

Programmers should write your automated acceptance tests

If you're a solo tester in an agile team, like me, you really have no choice but to have the programmers take responsibility for writing and maintaining automated acceptance tests. You'll be so busy with acceptance criteria, story and exploratory testing, you just won't have time.

The benefits of having the programmers in your team writing and maintaining these tests is that they will be maintained and executed as soon as any change occurs, so they'll be kept more up to date and less likely to go stale. They'll also be more useful in providing fast feedback to a programmer working on a change to a specific screen as the programmer can run the relevant acceptance tests, make a change and then ensure the acceptance tests still pass. If a tester were responsible for the automated acceptance tests, there is a lag between updating your application and your tests, which, if not properly managed, can lead of many tests repeatedly failing and spurious test results.

There will also be less tester vs programmer conflict as the

programmers will be able to identify regression bugs as they're created.

Programmers will have a better understanding of the architecture of your application and will be able to build testability features in so that the automated acceptance tests are more efficient and reliable.

Testers should write your automated acceptance tests

Software testers are particularly good at building automated acceptance tests that cover an end-to-end process in the system; often called *user journeys*. This is because they have a good understanding of the journey whereas a programmer may only understand the logic behind a particular screen. Testers should be involved in writing this style of acceptance tests so they are representative of real usage.

Some software testers are particularly good at knowing how to automate a web browser. Understanding how browsers work from an automation perspective is a highly developed skill that many testers have and having this skill leads to a more resilient automated acceptance test suite that wait for enough, but not too much, time for elements to appear and update.

Software testers are also very good at interpreting automated acceptance test results and investigating whether a bug exists or whether the tests need updating. If the tester is doing this work, then it makes sense for them to update the automated acceptance tests as needed.

What language should you use for your automated acceptance tests?

Use the programmer's language for your automated acceptance tests

Automated acceptance tests shouldn't be developed in isolation, so having these written in the same language as your application (usually C# or Java) will ensure that the programmers are fully engaged and will maximize the liklihood of having these tests maintained alongside your application code.

Even if the software testers are responsible for writing and maintaining the automated acceptance tests, having them in the same language the programmers use will mean that the programmers can provide support for any issues the testers have, and are more likely to collaborate with the testers on these. The testers also pick up knowledge of the language used for the core application which means they are more likely to be able to fix bugs that they find.

Strongly typed languages, like C# or Java, may at first seem daunting to software testers, because they're more verbose than dynamic languages, but they are actually surprisingly easy to learn due to the excellent support provided by IDEs such as

Visual Studio[55] or IntelliJIDEA[56].

Let the testers choose a language for your automated acceptance tests

If your software testers are responsible for writing and maintaining your automated acceptance tests then it makes sense to allow the testers to write these in whatever language they choose.

Dynamic scripting languages like Ruby[57] and Python[58] are particularly popular with testers as they are lightweight to install and easy to learn with an interactive prompt such as Interactive RuBy Shell (IRB)[59].

The benefit of a tester choosing a dynamic language like Ruby is that there are no licensing costs (unlike C# which requires Microsoft Visual Studio) and that means all testers have unconstrained access to these, as well as an unlimited number of build agents to run these tests as part of continuous integration.

As testers develop their skills in these languages they can quickly create *throwaway* scripts to perform repetitive setup tasks required for their story or exploratory testing: such as creating multiple records or rebuilding a database.

[55] http://www.microsoft.com/visualstudio/eng
[56] http://www.jetbrains.com/idea/
[57] http://en.wikipedia.org/wiki/Ruby_%28programming_language%29
[58] http://en.wikipedia.org/wiki/Python_%28programming_language%29
[59] http://en.wikipedia.org/wiki/Interactive_Ruby_Shell

Should you use the Given/When/Then format to specify automated acceptance tests?

You should use Given/When/Then scenarios to specify automated acceptance tests

The Given/When/Then format provides a high level domain specific language to specify the *intention* of automated acceptance tests separate to the *implementation* of your automated acceptance tests. This separation allows changing the test implementation method (eg. moving from testing the UI to testing a service) without changing the intention of the test and how it is written.

There are tools in nearly every programming language that allow you to specify tests this way: for example, Cucumber[60], SpecFlow[61] and JBehave[62] all use this format, and this format has become quite ubiquitous in automating acceptance tests.

[60] http://cukes.info

[61] http://www.specflow.org/specflownew/

[62] http://jbehave.org/

If the acceptance criteria on your user stories are specified in the Given/When/Then format then these are very easily transferred from a user story to an automated acceptance test.

You don't need the Given/When/Then format to specify automated acceptance tests

Writing automated tests in the Given/When/Then format creates an overhead of maintaining a collection of step definitions so that the plain language specifications are machine executable.

One of the biggest selling points of writing automated acceptance tests in the Given/When/Then format is that they are readable by non-technical members of your business. But in reality, business will seldom, if ever, read your Given/When/Then specifications, so it makes no sense to invest in the overhead required to implement these tests if they provide no extra communication benefit.

You are better off spending this effort on collaborating on non-executable story acceptance criteria that is fully understood between the business and your development team. These can be implemented however the team choose to do so without considering the need for business to access these.

Other domain specific frameworks such as RSpec[63] allow readable automated test specifications without the overhead associated with implementing the Given/When/Then format, and

[63] http://rspec.info/

these frameworks are often a better choice than a Given/When/Then based one when it is not needed.

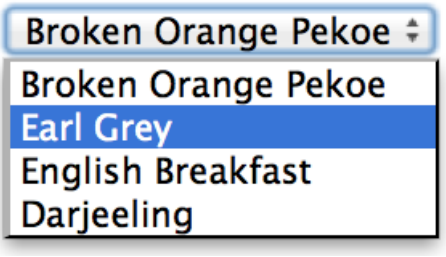
Should your element selectors be text or value based?

When writing automated acceptance tests for a web application, there are a couple of different ways of identifying and interacting with web elements, two of the most common are using *strings* or *values*.

Take this simple select list of tea:

```
<html>
<select id="tea">
<option value="bop">Broken Orange Pekoe</option>
<option value="eg">Earl Grey</option>
<option value="eb">English Breakfast</option>
<option value="darl">Darjeeling</option>
</select>
</html>
```

This renders as:



Select List Rendered

There are two ways to automate choosing something from this select list: by *string*, like "Broken Orange Pekoe" or by *value*, like 'bop'.

Use strings to interact with elements

The benefit of using strings to identify and interact with elements is that it's how a user uses your web application: a user will select "Darjeeling" from your drop down so why shouldn't your automated tests do the same.

If your web application is internationalized and you wish to run your automated tests in a different locale, you will also need to translate your selection as this will change with your locale that you set.

Modern JavaScript frameworks like Knockout[64] often don't generate values for select lists, so in this case you need to use strings to interact

[64] http://knockoutjs.com/

Use values to interact with elements

The benefit of using values to interact with elements is that values are the most resilient to change. For example, if "Earl Grey" was changed to "Earl Gray" and you are automating based upon value, then your automated acceptance tests will continue to work as they will use the value 'eg'.

This is also the best option if your web application is internationalized and you run your automated acceptance tests against a different locale: whilst "English Breakfast" may display "Petit Déjeuner Anglais" in French, it'll still happily be selected by using the value 'eb'.

Three non-contradictions

There's a few things I believe you can't contradict.

You can only grow by changing your mind

> "I would never die for my beliefs because I might be wrong."
>
> ~ Bertrand Russell

> "Those who cannot change their minds cannot change anything."
>
> ~ George Bernard Shaw

> "The measure of intelligence is the ability to change.
>
> ~ Albert Einstein

I used to believe that changing your mind was a sign of weakness, inconsistency; after all we're trained to hunt down inconsistencies as testers.

I now believe it's not only healthy to change your mind[65], but you can only grow as a person by changing your mind. Society is not accepting of this and the value of it is often overlooked.

What do you want to change your mind about today?

Everything is contextual

> "Taken out of context I must seem so strange."
>
> ~ Ani Defranco

Everything in life is contextual. What is okay in one context, makes no sense in another. I can swear to my mates, but never my Mum. Realizing the value of context will get you a long way.

You can always choose your reaction

> "Life is 10% what happens to you and 90% how you react to it."
>
> ~ Charles R. Swindoll

Probably the best thing I have ever learned in life is that no matter what life throws at you, no matter what people do to you

[65] I gave a lightning talk at the Test Automation Bazaar in Austin, Texas in 2012 titled 'I used to believe in the tooth fairy' that is about this topic.

or how they treat you, the only thing you can truly control is your response.

Once you understand that you are capable of controlling your destiny by choosing your reaction, it will set you free.

> "Today you are You, that is truer than true. There is no one alive who is Youer than You."
>
> ~ Dr. Seuss

All the best, may you wear your paradev title with pride.

Alister Scott

June 2013