The Future is Finding Zero Bugs (in QA)

Leslie Brooks (aka The BDD Coach)

Leslie@TheBDDCoach.com

Abstract

Depending on QA to catch bugs is inefficient – it is far cheaper if we never create the bugs – and it is far cheaper if we catch them before the Developers open a pull request. Depending on QA to catch bugs also means that QA takes the hit if requirements are late and the Developers take too long. The date doesn't move; QA gets crunched. Shifting all the way left - so far left that we prevent bugs from ever being written, and so far left that we deliver automated tests to the developers before they finish coding, eliminates these problems. BUT, the only way I know to do that is to use Specification by Example/BDD, and that is a methodology change that requires training and coaching, time, and buy-in from the Product Owners, Developers, and QA. I have tried it without that buy-in, but it doesn't work well. However, with that buy-in and the appropriate training and coaching I have seen a team deliver monthly releases for a year - and only two bugs into QA!

Biography

I have decades of experience in QA and specifically in automated testing, but I am always laser focused on value. Automated tests that cost too much to maintain don't deliver the value that we should demand. I love teaching Specification by Example/BDD; I love the tremendous improvements it can bring to the entire organization. When we are able to write better requirements, write better code (and fewer bugs), find and fix bugs faster, and deliver better quality code faster, then I feel an enormous sense of accomplishment.

1 Introduction

In the world we live in today, we expect developers to write code, and we expect that code to contain bugs. We expect the QA team to catch most of those bugs, but we expect that some of the bugs will escape into Production.

This is a world in which no one is truly happy – the Developers aren't happy because they spend too much time fixing bugs. The QA team isn't happy because they suffer from schedule cram-down – if the developers eat up too much of the schedule, the QA team is told they have less time to test. The customers aren't happy because they suffer with bad software, and the Product Owner isn't happy because the delivered software doesn't always do what they wanted it to.

There is a way to fix these problems; some companies are already doing it, but for most companies it is still in the future. This future is one where we truly 'shift left' and:

- The QA team doesn't run any tests (and therefore finds zero bugs)
- No bugs escape to Production

If we don't catch them in QA and they don't escape into Production, the only place left to prevent, catch, and fix them is in Development. In this future:

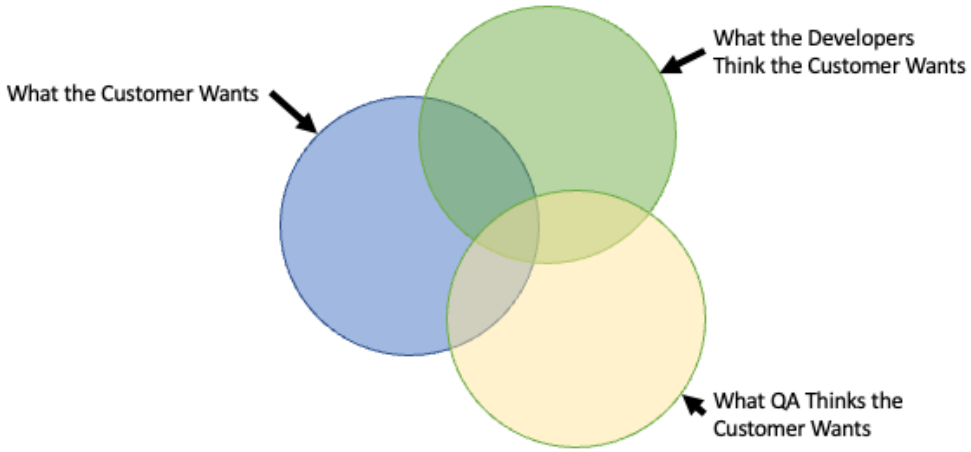
- The QA team delivers automated tests to the Developers before the Developers finish writing the code
- The Developers run the tests
- The Developers catch (and fix) all of the bugs before opening a pull request

This really is possible – there are companies doing it today.

2 How Does Our Requirements->Development->Test->Deliver Methodology Produce Bugs?

2.1 How Do Our Requirements Produce Bugs?

We write requirements so the developers will understand what the Product Owner wants to deliver to the customers. The developers read these requirements and say "OK; I understand what the customer wants, and I understand what I need to develop." The QA team reads these requirements and say "OK; I understand what we need to test" – and we wind up with this:



Of course this results in bugs. I like to talk about two kinds of bugs:

2.1.1 Little-'b' bugs

Little-b bugs are the ones where the code doesn't do what the developer intended – off-by-one errors are common examples. Other Little-b bugs could be that a database field isn't large enough for some of the data that we need to store in it, or that a phone number field accepts alpha characters, etc. Little-b bugs look like this:

Little-b bugs are a direct result of incomplete requirements or incomplete test cases. If we have comprehensive requirements then we can avoid creating Little-b bugs; if we have comprehensive tests that cover all of the corner cases and boundary conditions then we can avoid delivering them into Production.

2.1.2 Big-‘B’ Bugs

Big-‘B’ Bugs are the ones where the code does exactly what the developer intended; it just isn’t what the customer wanted. These can be very costly – the entire team may have spent an entire sprint – hopefully not two or three sprints – writing really good, clean, high-quality code that we can now throw away, because it doesn’t do what the customer wants! Big-B Bugs look like this:

Big-B Bugs are created when the developer's understanding of what the customer wants isn't the same as the Product Owner's understanding, and are a direct result of imprecise requirements.

2.2 How Do Our Developers Create Bugs?

Our developers don't want to create bugs – many developers are highly skilled, very careful in their coding, write really good unit tests, and use TDD (Test Driven Development). These all contribute to reducing the number of bugs that are created, but a typical developer still spends 30% to 40% of their time fixing bugs.

Developers create Little-b bugs by typing the wrong thing – entering the wrong field type when creating the database, or typing '<=' when they should have typed '<', etc. It is difficult to completely prevent these mistakes, but they can be caught if we have comprehensive tests.

Developers create Big-B Bugs by misunderstanding the requirements; we can prevent these bugs from ever being created by writing really clear requirements – and that is why Specification by Example/BDD was invented.

2.3 How Does Our Testing Fail to Find Bugs?

2.3.1 Incorrect Test Cases

If the test cases are wrong then we may run the tests but not find a bug. Test cases can be wrong because the QA team misunderstood the requirements; we can prevent this misunderstanding by writing really clear requirements. Test cases can also *become* wrong – they were correct when they were written, but the requirements have changed. When requirements change we are supposed to review the test cases that

map to that requirement – but the mapping could be wrong, or the person who reviews the test case might not make the right changes, or the review could be missed altogether. Specification by Example/BDD will definitely fix these process problems, because the requirements **are** the test cases. The test cases can only be wrong if the requirements are wrong!

2.3.2 Missing Test Cases

If we don't have all of the test cases we need – if we don't have comprehensive test cases – then bugs can also escape into Production. Specification by Example/BDD won't guarantee that these bugs are caught, but it definitely helps.

For example, if we have customer accounts with three different statuses – Active, Inactive, Suspended, etc. – then we would need to write three test cases to verify correct system behavior for each of those statuses. The Product Owner is very unlikely to review the test cases; if the test case for Suspended accounts is left out then it would simply not be tested.

Some teams would automate this with one data driven test case with all of the statuses in a spreadsheet. The Product Owner will almost certainly never review the spreadsheet, so if an account status is missed then it would simply not be tested. SBE says that the **requirement** must include the concrete examples; SBE doesn't allow it to be hidden away in a spreadsheet. Using Gherkin we would write a Scenario Outline that included all of the statuses; the PO, Developers, and QA would all review this requirement and it is very likely that someone would notice that a status was missing.

3 How Can We Eliminate Bugs?

Our current methodology – even a good Agile methodology:

- Allows us to create bugs
- Assumes that we will create bugs
- Assumes that we must have a QA team to catch the bugs
- Assumes that some bugs will escape into Production and be found later

If the problem lies in the methodology, then we can only fix it by changing the methodology. We can't fix it by giving our developers more training, or hiring more QA people, or switching to a better programming language or framework, or using a new QA tool.

3.1 How Can We Eliminate Bugs While Writing Requirements?

We can eliminate the Big-B Bugs by writing requirements using Specification By Example (SBE). SBE says that all requirements must include concrete examples, and must follow a clear 'Given these preconditions, When this event happens, Then this is the expected result' pattern. Both concrete examples and the clear Given, When, Then pattern help to eliminate (prevent) Big-B Bugs – it is much harder to misunderstand the requirements when they are written this way. (This Given, When, Then format is called 'Gherkin'.)

Using concrete examples also helps prevent Little-b bugs, because when we see concrete examples that leads us to think of other concrete examples. This helps us write comprehensive requirements – requirements that include all of the corner cases and boundary conditions.

3.2 How Can We Eliminate Bugs While Writing the Code?

3.2.1 Write Clearer Requirements

Using Specification by Example for writing requirements helps us write clearer requirements; it is much easier to understand what the customer really wants when the requirements include concrete examples.

3.2.2 Eliminate Incorrect and Missing Test Cases

In the world we live in today, eliminating incorrect test cases and making certain that our test cases cover all of the requirements, is very difficult. We must carefully read the requirements, think through the test cases required, and write all of the test cases. Writing test cases:

- Is extra work
- Takes extra time (and ensures that we *cannot* deliver automation in-sprint)
- Makes it possible for the QA team to misunderstand what the customer wants
- Makes for extra work when the requirements change – now test cases must be examined and possibly updated

The only way to effectively solve these problems is to not write any test cases – if we don't have any test cases, then we don't have any of the associated problems. Of course, we would also deliver all of the bugs into Production; that wouldn't be good!

However, if we use the SBE methodology and we do a good job of writing requirements up front, we eliminate all of these problems:

1) Our requirements and our test cases are one and the same; we don't have requirements and separate test cases, so we don't have to waste any time writing or maintaining separate test cases.
2) Our requirements must describe all of the features we plan to deliver, so we have a good start on completeness.
3) If we are careful to list all of the boundary conditions in our requirements, then we really can have complete requirements. Thinking of all of the boundary conditions is easier when requirements must include concrete examples.
4) If we always change the requirements before changing the code, then our test cases (the requirements themselves) are always up to date.

This doesn't mean that the QA team and the developers shouldn't *think about* test cases – they should. However, they should think of possible boundary conditions and design problems up front, so that the requirements can be fixed before the developers begin writing code. Getting correct and complete requirements before the developers begin coding means they will write fewer bugs, which means they will spend less time fixing bugs.

3.2.3 Deliver Automation In-Sprint

Rather than writing test cases, SBE says that we should directly automate our requirements. That is, the automation team should write an interpreter that treats our requirements as a formal grammar and runs them against our implementation[1]. If we do this:

- We don't waste time writing test cases
- The QA team can't misunderstand what the customer wants – we are literally running the requirements, not running test cases that we wrote based on our interpretation of the requirements.
- There are no test cases to be examined and possibly updated when the requirements change

If the Product Owner decides to change the requirements, and can make the changes without needing new requirements grammar, then changing the requirements does not result in any automation work for the QA team. If we do a good job of writing our requirements – if we write statements that can be easily re-used – then the Product Owner will be able to change existing requirements and even write many new requirements using existing grammar. When we re-use existing grammar – grammar that our interpreter already understands and knows how to automate – then the QA team has nothing to automate.

[1] Not everything should be automated; for example some graphical tests may be very difficult to automate but very easy for a human.

We won't deliver automation in-sprint for the first sprint, or probably for the first several sprints, but once we reach the point where the PO is using mostly existing grammar for writing new requirements, then we can consistently deliver in-sprint automation. When the PO can write twenty or thirty new requirements using existing grammar, and need just one new 'When' statement and one new 'Then' statement, then the QA team only has to automate those two statements to fully automate all of the new requirements.

Looking at this another way, in Programming 101 our professors taught us to make the upper layers of our programs as abstract as possible, and to push the implementation details as far down as possible. Test cases violate this rule in the worst way – our test cases are full of references to account numbers, passwords, names of text fields and database fields, URLs, and many other implementation details. When the requirements change we must make many, many test case and code changes. If we write requirements that talk about business purpose rather than about implementation, then:

1) The top layer of our program (our requirements) is more abstract, so we get lots of reuse out of our grammar and our automation
2) Implementation details have been pushed down to lower layers
   When requirements change, very little automation code has to change – and sometimes none at all!

3.2.4 Run the Automation Before Opening Pull Requests

Once we are consistently delivering automation in-sprint, the developers can run the requirements before they open a pull request. If any of the requirements fail, then there is a bug and they aren't ready to open the pull request. Once all of the requirements pass, then they can open the pull request, knowing that they haven't written or introduced any bugs. Of course the CI/CD pipeline should also run the requirements, either on every merge or nightly, just in case a developer forgot to do a fresh pull before running the requirements locally and opening a PR.

4 Where Has It Worked?

SBE is in use at many companies; some are using it well, but many are using it poorly. It is much more common in Europe than in the Americas, because it was invented in the U.K. and the early experts were all in Europe.

4.1 Case Study – BNP Paribas

BNP Paribas was agile, and had been agile for a number of years when they adopted SBE. They had very good metrics showing where their development teams spent their time; their developers were spending 34% of their time fixing defects.

They brought in a consultant who introduced two methodology changes – TDD and SBE. Nine months later their developers were spending 4% of their time fixing defects.

This change didn't happen overnight; it took training and lots of coaching and several months. SBE isn't a new tool, it is a methodology change and a radical paradigm shift, just as Waterfall to Agile was a methodology change and a paradigm shift. The whole team had to learn to think in this new paradigm before they saw the dramatic improvements that the methodology offers.[2]

4.2 Case Study – Raytheon Technologies

Raytheon brought me in to do SBE coaching and training for a data science team; the team consisted of data scientists on one side and developers on the other side, plus a QA team. The data scientists were the

[2] https://www.linkedin.com/pulse/case-study-bdd-improving-throughput-collaboration-simon-powers and https://cucumber.io/blog/podcast/bdd-in-banking/

customer; they told the developers what they needed and the developers worked to deliver it. Misunderstandings and bugs were common.

I began by training everyone in the new methodology and the new way of writing requirements. We began creating our grammar and our interpreter; there was zero automation when I arrived. As we automated more we had to do less manual testing, thus freeing up more time to write automation. Roughly nine months later we had a robust, reusable grammar and a robust interpreter; the data scientists could write requirements and they could be fully automated within a few days.

From that point bugs were essentially non-existent and the developers were able to shorten their release cycle to monthly releases. Over the next year they delivered into Production two defects – and everyone agreed that those two defects were not in the requirements. However, the developers said they should be counted as defects, saying "We should have known that the data scientists would want this". When the developers *claimed* a defect; I didn't argue!

We also had one release where the data scientists came back and said that we had delivered a bug. We looked at their bug, went back to the requirement, and said "This is the requirement, and when we run it it passes." The data scientists looked at the requirement and said "Oh… we wrote the requirement wrong; we will be more careful next time."

5 How Can We Do It?

5.1 Understand That SBE Is a New Methodology

SBE is not a new QA tool, nor is it a new Development tool; it is a methodology that requires buy-in from the whole team in order to work well. The team must include whoever writes the requirements in your company – Business Analysts, Product Owners, or Product Managers. Because of this, I teach my clients that they should absolutely ban use of the terms 'BDD' and 'test case'. I once had the manager over the BAs tell me "If you are writing test cases, do whatever you like, but don't involve us." I explained that we were writing requirements using the Specification by Example methodology, and that we had to do this in order to deliver automation in-sprint, and the CIO really wanted us to deliver automation in-sprint. At this point the BA Manager had to be involved – he couldn't allow us to write requirements without his team. Once we had his buy-in, then we could begin a real methodology and paradigm change.

The methodology differences are illustrated in these two RACI charts – in traditional Agile, only the Product Owner is responsible for the requirements. In the SBE Agile methodology the entire team – PO, Developers, and QAs – is responsible for the requirements. This **doesn't** mean that they must all sit together to write all of the requirements. It **does** mean that they must:

1) All agree on the language – the formal grammar – that they will use to write the requirements. If they are creating new grammar for an entirely new requirements domain, then they should probably all sit together to do this.
2) All agree on the definitions of terms that will be used in the requirements – for example, they should define what an 'Active' customer account means, and how it differs from an 'Inactive' or 'Suspended' customer account.
3) All review and accept the requirements. If the PO writes the requirements, the Developers and the QA team must review and accept them. If the QA team writes a requirement that they believe should have been included, the PO and the Developers must review it and approve it.

Agile

Tasks \ Roles	Product Owner	Developers	QA
Write Requirements	R,A	I	I
Write Code	I	R,A	I
Write Test Automation	I	I	R,A

Agile with SBE

Tasks \ Roles	Product Owner	Developers	QA
Write Requirements	R,A	R	R
Write Code	I	R,A	I
Write Test Automation	I	I	R,A

5.2 Bring In a Coach

A methodology change like this can be successful without a coach, but it can deliver value faster with a good coach involved. A good coach can help you avoid the common mistakes and deliver real value faster. I teach all of my classes “The first time we write Gherkin, we all write bad Gherkin. I know, because I did!”. A good coach can shorten the learning curve, quickly getting the team past the initial ‘bad Gherkin’ stage and to the stage where SBE is saving them time. Saving 30% of a developer’s time – perhaps $50,000 per developer per year – will pay for a coach in just a few months.

5.3 Do a Proof of Concept

Don’t try to move the entire company to SBE at once – start with just one or two teams, preferably teams that are starting on an entirely new product. Starting from scratch lets us write the requirements and build the automation in parallel with the code; attempting to convert an existing project to SBE is much harder. If we are starting from scratch everyone knows we have to write requirements, so it is relatively easy to write them using SBE. Then we can start automating long before the developers begin delivering anything to Production – and early automation delivers the greatest value to the developers.

5.4 Expand From There

Once the POC is successful, expand to a few more teams. Consider expanding to a team that works closely with the POC team; the POC team will be able to help bring them up the learning curve. Moving a couple of people from the POC team to the next team is also a great way to expand; you are seeding the new team with someone who already has experience. They have learned how to get through the hard parts and can share their knowledge with the new team.