CAPITA

Whitepaper

Behaviour Driven Development

zero known defect software releases

Challenging the assumption that "good enough" is really "good enough"

Behaviour Driven Development (BDD) is an Agile methodology which improves communication and collaboration between IT and the business, significantly increasing the likelihood that delivered software meets business requirements and expectations.

When the team uses BDD, is focused, and chooses to do so, they can achieve zero known defect releases. This is a positive, yet often underplayed, outcome of this approach to delivery.

This paper examines this causal effect of implementing BDD by first challenging the status quo with regards to defining software quality and then providing a worked mechanism to achieving zero known defect releases which also challenges commonly held beliefs regarding the need to prioritise and track defects.

When is software good enough for release?

This is a question that project teams will often ask during the process of software engineering. Equating "good software" to "meeting business requirements" is a reasonable place to start, but the definition of "meeting" a business requirement is then open for discussion and ultimately in many teams this leads to compromise.

This often results in a definition of having met a business requirement that includes a number of open defects still being present in the delivered software, each of which can potentially be a future headache for an organisation.

It may be that the acceptance criteria to exit from system testing (for example, a project following the V-Model) is defined in part as having no Severity 1 or 2 defects unresolved. This then implies that a number of lower severity defects may persist. Sometimes the proposed number of unresolved defects at lower severities (or priorities, if that is the selected measure) can be quite high. For example, test plans that permit the existence of 10 Severity 3, 20 Severity 4 and 40 Severity 5 defects are in effect stating that the requirements are considered met even when there are 70 problems with the delivered solution.

An argument may be made that these are fairly inconsequential on an individual level however they may amount as a whole, to a considerable failing in the delivery. If we consider a website where a single typographical error exists in production it is unlikely that this would be considered unacceptable. However, the same website containing 70 such obvious errors would, at best, make all concerned look unprofessional to their online customers.

At such levels of failure, disagreement can quickly occur: the acceptance criteria permit 70 low impact defects, but project sponsors will be fundamentally unhappy when they see this failure as a whole. In the worst case scenario they may refuse to sign-off against the delivered solution despite it technically having met requirements and acceptance criteria. In this way, the traditional prioritisation model for fixing defects fails: priority is set on a case-by-case basis, and does not usually take into account the total impact of outstanding defects.

We need a clearer understanding of what "good enough" means

1. The software should pass acceptance first time and be deployed to production without delay
2. The team should be proud of the quality of what they have delivered
3. The organisation should look forward to using the delivered solution
4. There should be no known defects outstanding

It is unlikely that many would contest the first three points, however the fourth is contentious. The perceived benefit of fixing a trivial defect versus the cost of the change may not appear to warrant it, so why bother? Project teams often hear their business sponsors state "it should just work!"

This is a reasonable expectation when purchasing goods and there is a definite competitive advantage to businesses that can provide an assurance that their delivered solution will "just work". For this reason it is proposed that all defects found, regardless of impact, should be fixed to enable zero known defect software releases.

Delivering zero known defect releases

It is important to differentiate between a release of software that contains "no" defects and one that contains "no known" defects. A zero-defect software release is highly unlikely. The complexity of modern solutions that integrate with numerous components, each of which has further interactions elsewhere typically prohibits this.

However, a zero known defect release is entirely possible. By this we mean that all scenarios that have been defined and executed should pass without failure and additionally all defects found via exploratory and non-functional testing should also have been resolved. This implies that the business sponsor has agreed that the scenarios are truly representative of the captured business processes. Also the scenarios demonstrate that the solution solves the business problems which were identified that were the cause of initiating the project in the first place.

Developing scenarios

Using BDD as our methodology the software engineering team (importantly, not just the business analyst, but the developer and tester as well) engage with stakeholders in conversations to ensure they understand the business problems and the agreed solutions.

We define problems as "Features" using the "As a… I want… so that…" syntax. For each Feature we produce one or more practical examples using the executable Gherkin syntax, "Given… When… Then…"

Specifying requirements through examples makes it easier to understand the intended solution. Effective, clear communication means that the sponsor can be confident that the scenarios properly reflect the features which solve the business problems. Ultimately, it is the initial communication that sets the scene for the engineering team to deliver a zero known defect release.

An Example

The Feature

As a Sales Executive

I want to be able to run a monthly report of sales totals by product line

So that I can present this information to the Board

The problem presented does not specify a technical solution

So we get clarification:

Given I am logged into the web reporting dashboard

When I execute a report for a selected month

Then I should be presented with a downloadable PDF

And the PDF should contain a single line item for each product line

And each line item should contain the product line name and the sales total

We have found that a practical approach to reduce complexity is to keep the number of "Given" steps down by wrapping up functional steps, such as with "Given I am logged in" which does not specify either a username or password, nor does it state which fields to complete and that a login button should be clicked. Likewise we keep the "Then" clauses to a small number, representing those important facets of the behaviour we intend to satisfy.

Manual regression testing

So when is "good enough" really "good enough"? For the solution to pass acceptance first time we must be confident right through development that new code does not break previously written code. Otherwise we reach acceptance with a proportion of the solution that has not been subject to a recent regression test.

In Scrum this would require that in Sprint 2 we not only test the deliverables for that Sprint, but additionally regression test Sprint 1. Attempting to do this manually is clearly time-consuming, and will quickly result in insurmountable test debt.

The time to manually regression test (M) in any given Sprint is broadly equivalent to:

M = (f x (n -1)) x R% where M is the time required for manual testing, f is the volume of work achieved in each Sprint (for simplicity we are assuming that this is constant across Sprints), n is the current Sprint and R% is the percentage of functionality to be regression tested.

For example:

If we are on the third Sprint, and we want to regression test the entirety of Sprints 1 and 2:

M = (f x (3-1)) x 100%

For simplicity let us assume that 50% of each two week Sprint is devoted to testing:

M = (10/2 x 2) x 100%

ie: it will take 10 days additional to Sprint 3 to regression test all that was provided in Sprints 1 and 2.

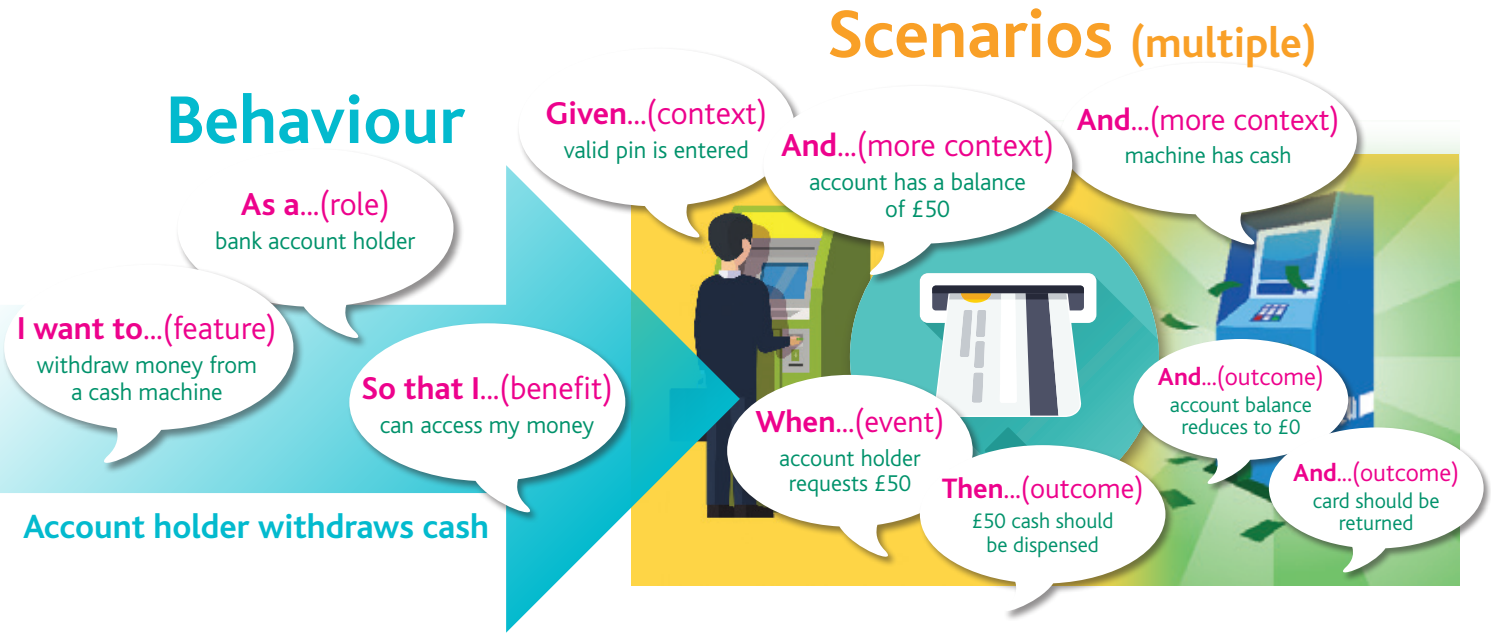
Clearly this is unachievable as Sprint 3 is itself only two weeks in duration. This means that to regression test everything from Sprints 1 and 2 we would not be able to undertake any new work in Sprint 3.

Is there a compromise that will work?

We can spend only 2 days regression testing Sprints 1 and 2 ie. M = 10 x 20%

This has two consequences:

- The time available for Sprint 3 is reduced by 2 days
- 80% of functionality delivered in Sprints 1 and 2 is not regression tested in Sprint 3



Partial mitigation can of course be made by varying the 20% of functionality that is tested in each successive Sprint so that in theory, over time, all functionality is covered. However in practice this is impossible to achieve as with Sprint 6 and n-1 equalling 5 we will have returned to a situation of 100% of the current Sprint being required to regression test just 20% of previous Sprints. We therefore have to reduce R each Sprint, so it is impossible to ever fully regression test the solution using manual techniques as we are defining an asymptotic curve.

Following this approach, a manual model of testing clearly cannot provide assurance at the start of the acceptance phase for all bar a small subset of the delivered features.

The team should regularly ask:

> **Are we producing software that helps solve the business problem(s)?**

Using automation; providing continuous assurance with BDD

Instead, when following BDD we can provide continuous assurance, using automation in the form of continuous integration, coupled with demonstrations and timely feedback to the sponsor.

To ensure that we are always moving in the right direction the team should regularly ask: "are we producing software that helps solve the business problem(s)?" If the answer is "no" then an immediate halt should be called and the team and sponsor need to engage in focussed collaboration to get back on track. The situation if the answer is "maybe" also demonstrates a divergence from the solution meeting the business problem.

This could be because the problem the team is attempting to solve remains poorly defined or because the testers are working behind, not alongside the developers, leading to a situation of reactive defect fixing, rather than proactive delivery of a working solution.

The EuroSTAR 2014 Report into the Practices and Attitudes in Software Testing notes that only 48% of survey respondents stated that test automation is carried out during development "all or most of the time".

For most respondents, some degree of automation occurs at some point in the Software Development Lifecycle (SDLC). This means that over half of software referenced in the survey is not subjected to continuous test automation at the point of development which probably moves those projects into the "maybe" category in relation to solving the business problem.

Even more concerning is that according to the survey only 30% of organisations start their test automation during the requirements or design phases. This is precisely where the plain language of BDD can aid communication and requirements gathering.

Continuous integration with BDD Scenarios immediately highlights to the engineering team when regressions occur. Historically, the process of prioritisation then dictates when these can be fixed. Instead we believe that three changes can be made to the defect tracking process:

1. **No defect is assigned a priority**
2. **No defect is recorded in a separate defect tracking tool**
3. **All defects are fixed**

For the third point we need practical guidance to steer the day to day work of the engineering team to achieve this goal, however to ensure buy-in it is important that working towards zero known defect releases is a choice of the team, and not seen as a management edict. The following three rules have proven successful in steering the team to a zero known defect release:

1. **The team will attempt to fix every outstanding known defect by the end of each day**
2. **The team guarantees to fix every outstanding known defect by the end of the working week and always before delivery**
3. **The team never allows the open defect count to go above three at any one time**

CAPITA

Whitepaper

The impacts on tools and process

Changing the engineering team focus around from developing and fixing to delivery certainty means changing business perceptions. To obtain working software with zero known defects, consideration needs to be given to the level of reporting that is required to support this activity by ensuring only the key metrics are reported which are truly useful to delivery.

This can be a challenge as it means dispensing with some elements that have previously been seen as key, such as defect tracking tools. BDD scenarios self-track: they're red when they fail and go green again when they pass. As the team becomes fluent in the conversations of BDD, the number of defects found per release approaches zero and the use of tracking tools often become redundant. For the small number of issues found during exploratory testing, outside of BDD, the mindset of the team, following the above three rules, becomes one where-by defects are resolved almost immediately. Hence, the value of a defect tracking tool for non-BDD defects is also questionable.

It is easy to think that if no defects are found then the need for dedicated test resource is removed. This is wholly incorrect and will result in a rapid escalation of the defect count. Instead of reacting, finding defects post-development, the testers are heavily engaged working alongside the business analysts and developers to ensure no defects ever leave development. The value of a tester working proactively to prevent defects is much greater than reactive testing as the cost to fix issues when they are found earlier is much reduced.

Reporting that no defects are found is a good thing providing that analysis and scenario construction was rigorous. This gives the sponsor confidence that the software solution genuinely works, and works as they intended.

What does success look like?

For this model to be successful, the sponsor needs to have both implicit and explicit trust in the engineering team to deliver. From our experience we see that as confidence increases so the volume and frequency of metrics requested generally reduces.

Teams need to be focussed on the quality of what they deliver. Implementing BDD provides a mechanism to achieve this. The quality of a delivery should be viewed as more important than the quantity of what is delivered to encourage delivery certainty. Take the example of an operating system update that provides new features. End users will be much more accepting of a new release with a smaller additional feature set that does not break existing functionality than a release containing lots of change, but at the expense of destabilising pre-existing functionality.

If we take a look at our criteria for what we believe is "good enough" at the beginning of this paper it includes delivering a solution with no known defects. We believe that using BDD enables us to deliver on this and make sure that the solution does what it promises ensuring that the business gets the best possible outcome on time and to budget.

Author
Colin Deady, Test Manager

Key success criteria for BDD

- Involve all stakeholders at each stage
- Try to keep scenarios as simple as possible without repeating syntax where possible
- Raise the possibility of zero known defect releases with the team – can they buy in to this? *Remember: it needs to be a choice of the team*
- Implement continuous integration to give rapid feedback to Development
- Use executing scenarios as your regression test suite

For further information about these services please contact:

Email: **marketing.itps@capita.co.uk**
Tel: **+44 (0) 8456 077466**
www.capita.co.uk/itprofessionalservices

Follow us: @CapitaITPS Capita ITPS Capita ITPS Capita IT Professional Services