The Beginners Guide to BDD

Konstantin Kudryashov
Oct 07, 2015
https://inviqa.com/blog/bdd-guide

Introduction

When launching a new digital project, there can be a disconnect between:

- The business being truly able to define the desired outcomes
- The developer's understanding of what needs to be built, and
- The business' understanding of the technical challenges their requirements may present.

Behaviour-driven development (or behavior-driven development) can help achieve all of the above and ultimately, helps a business and its technical team deliver software that fulfils business goals.

'The Beginner's Guide to BDD' is an introduction to behaviour-driven development – an approach to development that improves communication between business and technical teams to create software with business value.

This guide is for both technical and business professionals to learn about BDD, how it can benefit projects of all sizes, and how to implement it with confidence. It is structured to reflect the flow of the BDD process.

For more information on how Inviqa itself introduced BDD, check out the 'How we use BDD at Inviqa' section, or contact us to learn how Inviqa can help your organisation or technical team adopt BDD.

Contributors to this BDD guide

Konstantin Kudryashov, former BDD practice manager

Konstantin headed-up Inviqa's BDD (Behaviour Driven Development) practice. To ensure software development meets business goals, Konstantin established the BDD pipeline, a set of processes and tools that aid collaboration between technical and non-technical teams.

Alistair Stead, former CTO

Alistair helped bring BDD principles into Inviqa and continued to support the delivery of projects and the progression of BDD practices within the development teams.

Dan North, Dan North & Associates

Dan is the originator of BDD and is a regular keynote speaker at conferences, covering topics from software engineering to Agile methodologies. Dan owns and runs his own consultancy, Dan North & Associates, where he trains and coaches on software delivery and organisational change.

What is it?

Software development is often deluged by overwork, translating directly into wasted time and resource for both engineers and business professionals. Communication between both parties is often the bottleneck to project progress when engineers often misunderstand what the business truly needs from its software, and business professionals misunderstand the capabilities of their technical team.

Shrouded in miscommunication and complexity, the end product is often technically functional, but fails to meet the exact requirements of the business. Behaviour-driven development (BDD) aims to change this.

BDD is a process designed to aid the management and the delivery of software development projects by improving communication between engineers and business professionals. In so doing, BDD ensures all development projects remain focused on delivering what the business actually needs while meeting all requirements of the user.

Key benefits of BDD

- All development work can be traced back directly to business objectives.
- Software development meets user need. Satisfied users = good business.
- Efficient prioritisation - business-critical features are delivered first.
- All parties have a shared understanding of the project and can be involved in the communication.
- A shared language ensures everyone (technical or not) has thorough visibility into the project's progression.
- Resulting software design that matches existing and supports upcoming business needs.
- Improved quality code reducing costs of maintenance and minimising project risk.

The BDD approach can largely be divided into two main parts. The first is the practice of using examples written in ubiquitous language to illustrate behaviours (how users will interact with the product).
The second part is the practice of using those examples as the basis of automated tests. As well as checking functionality for the user, this ensures the system works as defined by the business throughout the project lifetime.

In BDD, business value and user requirement are never secondary during development.

The concept of BDD was introduced by Dan North, an experienced technology and organisational consultant based in London, as a means of clearing confusion between testers, developers, and business people. To learn more about the origins of BDD, check out our interview with Dan North, or watch the video below of Konstantin Kudryashov introducing the fundamentals of BDD.

Starting with a goal
At the highest level, software development is just a way to support human behaviours by automating, replacing, or creating them. Every behaviour change should help the business or the company to achieve its goals, whether that's increasing revenues, improving usage or reducing waste.

In BDD, projects are delivered in sets of capabilities. Capabilities are essentially features in software development that enable users to be more efficient. For example, automating the ordering process of an ecommerce store or automating warehouse management.
In order to meet business requirements, you need to understand exactly what you want the software to achieve from a business perspective. In BDD, this then becomes your business goal: the driving force behind the project.

A good business goal is specific, measurable, attainable, relevant and time-bound (SMART).
Here are two examples:

Bad example:
We are going to generate more revenue by making more sales

Good example:
We are planning to achieve a 15% increase in revenue through our online channels in 6 months

Having a SMART goal not only helps to focus the project, but allows the impact of the project to be measured. This ensures that any and all development can be linked back to the business goal and ensures the end product meets business requirements.

Impact mapping
Once the short-term goal for the project has been defined and the timeline agreed, you can begin planning how to achieve your aim.

In software development, it is all too common to try to achieve goals by adding new features to applications. Delivering features for the sake of delivering features is a common problem in software development, and is the reason behind many of the most infamous project failures. Instead of trying to deliver as many features as possible, you need to identify how the features will support the users behaviours or the business.

Although any piece of software consists of hundreds of features, it is important to understand that not all of these features need the same level of attention, care, or even development practice, and it can be easy to spread the development team's attention too thinly across an entire backlog.

The increasingly popular Agile practice for managing the project roadmap and laying out all potential features is called Impact Mapping.

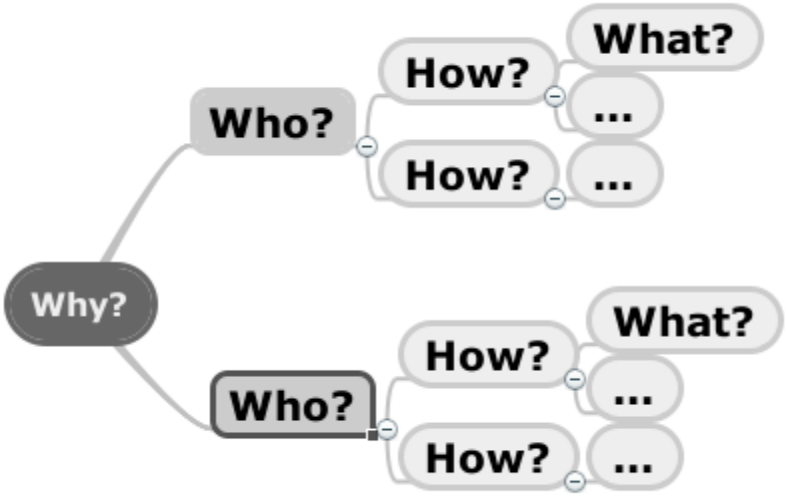
Impact Mapping is a technique that helps you to outline all of the alternative ways to reach your decided goal and works by analysing users' behaviour. We assume that software cannot achieve business goals on its own; the only way software can get us closer to the goal is by supporting particular human behaviours.

Impact Mapping is a technique that helps you to outline all the alternative ways to reach your decided goal by analysing users' behaviour. We assume that software cannot achieve business goals on its own. The only way software can get us closer to the goal is by supporting particular human behaviours.

For example, an online shop doesn't bring you money on its own; people bring you money by purchasing products from it. Similarly, better system logging doesn't improve performance of your website on its own; technical specialists using it do. Looking at software through a prism of those who use it is a very valuable way to plan and direct the project delivery.

Impact Mapping is a very simple technique built on top of conventional mind mapping, with a slight spin on it. An impact map has four distinct levels consisting of the business goal, one or more actors, one or more impacts and one or more ways to support or prevent these impacts.

You always start with the business goal; it is your map's root node. You then grow the map out from the goal by first identifying all the actors (e.g. the customer or the team) that could help or prevent the project from achieving the goal. Each actor could have multiple ways to help or hinder achieving the goal, we call these impacts.
The last layer of an impact map defines what the project or delivery team can do in order to support or prevent particular impacts from happening, and this is the layer where your software options come into play.



Value and complexity analysis
In collaboration-heavy Agile methodologies like BDD, it is very important to clearly distinguish and agree priorities. In behaviour-driven development this is done by using two techniques: value, and complexity analysis. Value analysis helps us to quickly identify low-cost, high-value features in the backlog. Complexity analysis helps us to choose the right development and collaboration approach to the project as a whole, as well as each individual feature.

One of the most interesting complexity analysis frameworks BDD practitioners use is called Cynefin. Cynefin is a sense-making framework, which means that it helps you to make sense of the problem at hand so that you can act accordingly. Dave Snowden made this video which broadly introduces Cynefin.

In BDD we use Cynefin to identify which features require the most attention and collaboration with stakeholders as well as which features we can reuse from other open source libraries or even outsource to specialised companies. We also use Cynefin to make strategic planning decisions based on value/complexity analysis that helps us easily identify commodities (features we work on regularly) and differentiators (new areas). Cynefin helps to ensure that you're using the right examples.

Liz Keogh did an excellent presentation on BDD and Cynefin at one of Inviqa's meetup events.

Planning in examples
Imagine that you could describe the most complex of ideas to somebody with a different skills-set without introducing misunderstanding or the wrong expectations.

Misunderstanding and wrong expectations are the primary reason behind overwork and waste in the software industry. Engineers often misunderstand what a business actually wants or needs and business people often misunderstand what engineers are actually capable of. This all contributes to an environment where the technical team continuously delivers, but does not meet the needs of the business and can miss a number of opportunities. This was always a serious problem, but the growth of Agile has made it much more apparent.

Avoiding misunderstanding is a challenge that is not limited to software development. As humans we have encountered this problem throughout our history; every time people communicate, there is a risk of misunderstanding. The difference with non-software communications is that we have had literally millions of years to come up with a solution. And we did. What is the easiest way to remove misunderstanding? Using examples.

Understanding through examples is one of the earliest learning practices we start with as children to remove ambiguity and provide context to what we are trying to understand. So why can't we just use examples to remove ambiguity from our software discussion? As a matter of fact, we can.

Imagine we have been given a task to add an 'include VAT and delivery cost to the total price of the customer basket' feature to an existing ecommerce project with following set of business rules:

> VAT is 20%
> Delivery cost for small basket (less than £20) is £3
> Delivery cost for medium basket (more than £20) is £2

Can you use this information to deliver a working feature? Of course you can! But what are the chances that you will get it wrong or that you have not fully understood what these rules mean? For example, these rules do not specify whether to add VAT before delivery cost to the basket total or after. How should you handle a situation where the basket delivery cost is exactly £20? What happens if there are multiple products in the basket?

None of these questions are fully answered in the business rules or the feature description itself which creates ambiguity, and therefore misunderstanding. But what if, in addition to these three business rules, we provide a set of very simple examples on how the application will behave with them?

For example:
Given a product is priced at £15, when I add it to the basket, then the total basket price should be £21
Given a product is priced at £25, when I add it to the basket, then the total basket price should be £32
Given a product is priced at £20, when I add it to the basket, then the total basket price should be £26

When written in this way, the ambiguity around total basket price is completely removed. That's the beauty of examples. They tackle uncertainty by illustrating business rules in action. In a nutshell, if you now convert these three examples into tests and use those to drive development, you're doing BDD.

Usage-centered design
Examples illustrate human behaviours, but whose behaviour are we trying to illustrate? One of the biggest promises of BDD is to deliver software that matters. In order to achieve this you need to understand who exactly will use it and why.

Much software is designed and built with little consideration for how it will be used and how it can best support the work its users will be doing.

Usage-centered design is an approach to software development where every single bit of deliverable functionality focuses on the intentions and patterns of users. Users are grouped into so-called 'abstract actors', which define different roles involved in the software interaction.

For example, returning to our ecommerce example, imagine that we want to rework the catalogue. Instead of jumping straight away into the implementation stage, we try to identify who the audience is for the feature. In a short discussion with the product owner and analytics team we discover that most customers on the platform look for a very specific product, but try to search for it on the site using ambiguous terms.

Being unable to find the product via search, they look and find the product using the catalogue navigation, which is also less than optimal. After further investigation we identify that the actual user of this feature is a 'customer interested in specific product'. The feature request then changes from a large task of 'rework the catalogue' into the simple 'add support for ambiguous terms in product search'.

Focusing on users behaviour is the key concept behind usage-centered design and one of the most important concepts in BDD. In BDD we not only discuss examples of human behaviours and interactions, but we also discuss them from the point of view of their users. By focusing on the users and their behaviours, we understand who the system is built for and as a result, deliver software that meets the needs of all who use it.

By analysing behaviours of people that will use the system, we constantly analyse different options and alternatives that would best meet their needs. In doing so, we are opened to different implementations and optimisations. Caring about behaviors not only helps us to deliver the software itself, but build the technical system to directly meet the needs of users.

Ubiquitous language
It is really hard, or almost impossible, to form a two-way communication between groups of people that operate in different business spheres without stumbling upon situations where both sides use the same language and even words, but imply a different meaning.

For example, if building a banking application, you might be asked to implement an 'account' feature in the next iteration. If you don't have enough experience with this particular business or company, you may assume

that this feature relates to ‘user accounts’ or ‘user management’ whereas it may refer to the introduction of ‘credit/debit accounts’ into the system.

If this misunderstanding isn’t cleared in time by the development team, they could spend anywhere from a couple of weeks (using Agile) to years (using Waterfall) building an ‘account system’ that the client didn’t actually want or need. This problem is generally known as ‘the feedback delay’ or, in this particular case, ‘the cost of translation’.

BDD attempts to eliminate the cost of translation by reducing the size of feedback loops and enforcing the example-based conversation about each feature before they are built. But just asking for examples is not enough, because different sides will use different language to produce examples, eventually causing discrepancies. The solution for this problem is the concept the BDD community borrowed from DDD (Domain Driven Design): ubiquitous language.

Ubiquitous language is a language created by developers and the business in order to discuss features and talk effectively in examples. It is not a technical language, but it is not a business language either; it is a combination of both. The key in the two sides becoming effective for joint communication lies in their ability to accept, to a certain extent, the points of view and understanding of each other. Ubiquitous language is an integral part of BDD. It is almost impossible to be effective with examples without striving for a shared language and understanding.

Examples could take a different form depending on the business and the nature of the ubiquitous language it creates. For example, the stock or banking business domains will most likely use some form of a spreadsheet to provide examples for business processes or important formulas.

In the software domain, people tend to use diagrams and technical maps. That said, the most common and adopted format for examples is Given-When-Then which is the format you’ll see most used by the BDD community. It follows a very simple structural agreement.

'Given, describes the initial context for the example
'When' describes the action that the actor in the system or stakeholder performs
'Then' describes the expected outcome of that action

The primary reason why this format is so popular is because it is the natural way people speak. In many cases, when we ask our clients to provide examples of how they expect a particular capability to be delivered, they naturally describe it in a sentence such as: ‘when I do that, then this should happen’. Here’s an example of an example:

Feature: Returned items go back to stock
In order to keep track of stock
As a store owner
I want to add items back to stock when they are returned

Example: Refunded items should be returned to stock
Given a customer previously bought a black sweater from me
And I currently have three black sweaters left in stock
When he returns the sweater for a refund
Then I should have four black sweaters in stock

Example: Replaced items should be returned to stock
Given that a customer buys a blue garment
And I have two blue garments in stock
And three black garments in stock
When he returns the garment for a replacement in black
Then I should have three blue garments in stock
And two black garments in stock

This user-story identifies a stakeholder, a business effect, and a business value. It also describes several scenarios, each with a precondition, a trigger, and an expected outcome. Each of these parts is identified by the more formal part of the language (the term 'Given' might be considered a keyword, for example) and followed by a language that the development team and business created together. In the above example, the ubiquitous language includes terms like 'stock', 'replacement' and 'returns'.

The formality of the language used in examples also plays an interesting part in the development process. The example above has an obvious structure that is very easy to follow or break down. That makes it an easy target for automation.

Introducing the three amigos
One of the most common mistakes that teams commit when starting to work with examples is to assume that writing down these examples is the most important part of the process, and that giving this responsibility to a single person is enough.

Behaviour-driven development is a highly collaborative process. Talking in examples requires not only more than one person to have that conversation, but requires different perspectives and experiences in order to be efficient. One of the core ideas behind BDD is that no single person has the full answer to the problem. This is where the idea of The Three Amigos come into play; all angles can be covered if you bring together the insights from a business person, a developer and a tester.

Business roles such as a product owner or business analyst remain in the business value and risk evaluation domain; their insight and input is extremely important if you want to deliver software that matters (these people will generally ask questions like 'is it valuable? Do we really care about this?').

Developers, in the most part, remain in the so-called 'solution' space. Their input will give you an invaluable context into how much detail the feature should have in order to be implementable.
Testers usually represent the 'problem' space; they see flaws happening in the system even before development commences.

The core premise of BDD is not to replace these different expertises, but to move them all in one room and have a meaningful discussion about each feature before development starts. This process is called the Three Amigos or an Example Workshop and must be conducted before the team or a developer commits to deliver the feature in question.

Development through examples
As discussed in the previous section, examples in (semi) formal language are naturally easy to automate. This means that you can feed an example to an automation tool (such as Behat, Cucumber or SpecFlow) that will transform it into the automated specification. Developers can then use this automated specification to drive the application development.

As a matter of fact, the language we used in the example of the previous section has a name: Gherkin. Gherkin is a keyword-based, line-oriented language. This means that, except when you need to break sentences down

into lines and use special reserved keywords, you are free to use actual ubiquitous language to write down behaviour examples.

This simpler format also makes it easy to support Gherkin in different programming languages. At the time of writing, there are Gherkin automation tools written for Ruby, Python, PHP, .Net, JS, Java and its derivatives.

All automation tools work following the same pattern. They take the feature file with a scenario:

> **Scenario:** Refunded items should be returned to stock
> *Given* a customer previously bought a black sweater from me and I currently have three black sweaters left in stock
> *When* he returns the sweater for a refund
> Then I should have four black sweaters in stock

And using simple text matching algorithms associates each step inside the scenario with a programming instruction that automatically checks the step premise by simulating the described behaviour. This creates an executable specification, which can help developers both guide the development of a feature and make sure that this feature stays working after it is deployed, rendering an effective regression mechanism.

The BDD loops

Independent of the platform, technology or language you use to deliver your application, one thing is always constant: in order to succeed, every application needs to constantly adapt. As the proverb goes, change is the only constant. Markets change, technologies change, our understanding of business changes and we get new opportunities every day. So the way we deliver software should reflect this ongoing process of change.
In order to support the pace of change we face in most modern projects, adopting scenario-based test-driven development is not enough. Scenarios tend to describe the behaviour of the application as a whole, on a higher, modular level.

But, in order to support the slow growth and change of these modules, we need to be able to do drastic and much faster changes in the system underneath them.

Simply put, when optimising a car for continuous performance improvements, knowing that it can still drive is not enough, you need to be able to observe and impact individual characteristics or elements of how it drives. And that's what unit testing is in BDD: a way to support big, visible changes in the application behaviour by being able to do a lot of small (sometimes invisible), but very stable changes inside it.

Unit tests are small pieces of code, which have a sole purpose to evaluate a small isolated part of the system. In BDD, we don't call unit-tests tests, we call them object specifications and we treat them as examples of how small isolated parts of the system should behave, rather than a way to test them. In BDD, we acknowledge that in order to support the ability for systems to change, we should be able to safely make big changes (supported by automated scenarios), as well as the small ones (supported by automated object specifications).

In full-stack BDD we intertwine both the scenario and object specification writing process together with the development effort so that it creates a coherent delivery process – where we constantly deliver software of required quality, both on the outside and on the inside.

Taking a scenario from the previous section as an example, we would start from automating it against high-level modules or even a user interface of the application being built. We will then execute this scenario step by step, trying to make these steps pass. Some steps will require us to simply fine-tune the application configuration to make them pass. Others will require us to change how the underlying object model works - that's when we'll go down to the object level and will start writing missing object specifications or describe additional behaviour to the existing ones.

Following the full-stack BDD process, we end up creating a software solution that is not only well-built and tested, but also what the business needs. Every line of the production code, configuration or even specification is directly linked to its particular scenario, which itself is linked to the ultimate project business goal. Thus, we end up knowing how much business value each line of our code must deliver. And if it doesn't as a result of market or business reasons, we know exactly which lines of code we need to replace or even remove. This is the way BDD ensures it constantly delivers software that matters.
Q&A

In order to provide some added context on BDD, we interviewed the founder of BDD, Dan North on why he introduced BDD, the pain points he was trying to address, the core of BDD, and its biggest impediments.

About Dan North
Dan North is world-renowned for his work with Agile methodologies, including the introduction of behaviour-driven development. Dan has been providing coaching / training and consulting services for over 20 years and he is currently the Principal Consultant at Dan North & Associates. You can find out more about Dan here: dannorth.net/about/

What was the biggest pain-point you were looking to address when you introduced BDD?
When I introduced BDD it was a far smaller thing than it is now, so the definition has evolved a lot over time. When I first started using the term 'behaviour-driven development' I was coaching and it was in the early 2000s. I was working with developers, teaching them TDD and the XP method whilst working for ThoughtWorks, who were very much pioneers and early adopters of Agile methods.

I kept running into the same arguments, where programmers would say things like 'hang on, we don't write tests – we have testers for that'. And I'd say 'this isn't really a test, you're running the test as a client to your code'. They'd then respond again with 'Yes, that's a test'.

On the flip side to this, the testers weren't happy that the programmers were writing tests, because they didn't believe that they were qualified or capable. There was also a slight nervousness that they'd also be out of a job.

So it was becoming clear that there was a lot of contention around the word 'test' and the principles of test-driven development (TDD), had almost nothing to do with testing and was more about small scale design. To sidestep the confusion I just essentially changed the words. I was a fan of NLP (neuro-linguistic programming), which then broadened into general behavioural psychology, so I thought I'd try a linguistic exercise to drop the word 'test' entirely. This lead me to think, 'if I'm not using the word "test" anywhere, which word should I use?'

I started saying: we're not writing tests for this, we're writing examples. We were trying to specify what the code needed to do ahead of time and make that code executable – so we were trying to describe the behavior of the code before it actually existed and then make the application code behave in that way.

You can get quite a long way without using the word 'test'. When I first started saying to programmers 'we're not going to be writing any tests; instead, we're going to write examples around the use of that API instead', they responded positively. We then discussed writing a spec and writing code, so we could see if it was executable (which would allow us to prove that it works) and they were very positive about this shift. Suddenly, adoption quickly increased.

The core problem I was trying to solve really was skepticism and a reluctance to adopt practices as useful as TDD, held back by its vocabulary which was largely due to the words, rather than the actual principles.

Why does BDD work?
Once you move away from the contentious vocabulary, you can use the word 'test' to actually refer to testing. Suddenly, once you've stopped using the word 'test' to mean API, or example etc., you discover the tester's role is right at the core of the process. They were sidelined before and are now being brought into the middle.

I remember speaking to business analyst Chris Matts and he asked me to explain BDD. I started explaining it to him and describing how to use examples, set expectations etc. and he said it sounded just like analysis.

We decided to take it to the next level of abstraction, which was when it moved towards the acceptance testing space – moving towards the next level of projects, where you describe scenarios, features, stories etc – so you had BDD at two different levels. Because BDD was very similar at both levels, I could consider whether something was a feature-level example, which we call scenarios, or whether it was a code-level example, which we call examples.

The point of BDD is to get people communicating and bridge the gap between 'technical' and 'business' people – two words that I'm using deliberately because that's how Kent Beck (creator of XP) originally described his motivation for XP.

Rather than going into detail on the principles and process, he said 'I'm trying to get business people and technical people talking to each other'. That's what BDD does; it causes you to have these conversations as you're fleshing out the scenarios and lots of gaps surface – which were there anyway, but they would have surfaced when it's too late to do anything about them.

So what BDD does is help to surface these problems a lot sooner. Because of this, people find the analysis piece very useful – and then the fact you can then take the analysis and drive it into the code, gives you a whole holistic way of describing behavior and being confident about the behavior of your application.

What is the biggest impediment to BDD?
So, you think of BDD as a way of using examples and conversion to drive scenarios to the surface in order to deliver software that matters. If you are structurally or organisationally impeded, BDD may not be as effective. For example, it could be a geographic limitation thing, where you have people in different offices, or you could be held back by organisational boundaries, such as offshore testing.

A lot of companies offshore their testing, which is crazy. They're separating by roles rather than business area. So if I have a completely self-contained, autonomous team – they can be anywhere in the world and get good work done.

What I'm trying to do is to have one conversation with someone with deep technical knowledge, someone with deep business knowledge, and then a professional tester. Getting these guys discussing the domain, discussing the problems and talking about how they can resolve them will lead to them surfacing all kinds of assumptions and disconnects. Anything that impedes this is going to make it work less well.

Sometimes the challenge is not structural – it could be social. For example, there has traditionally been a disconnect between business analysts and testers or it could be that the developers think that the testing can be done via offshore resource and they can just send a specification, which in my experience doesn't work.

How we use it
Inviqa originally started working with BDD principles back in 2012. Initially, we didn't adopt any automation, but solely used BDD for requirements gathering and documentation, using Specification by Example as the starting point.

Inviqa's adoption of BDD was born out of a necessity to address a number of common friction points in projects, such as functionality defects, misunderstandings and misinterpretation in implementation. We also needed a way to prioritise the backlog, as well as a tool to help us facilitate the conversation with the client.

The transition to BDD was straightforward. We didn't use it on every project but started with larger, more complex projects where the benefit would be greater. It took around 12-18 months before we fully integrated the process into all of our projects, which was around the time that Konstantin joined the team. Once Konstantin had settled in, we accelerated our adoption of BDD and also the automation side of things, which focused on automating the conversation piece that we were doing manually before.

The structure of our development teams didn't change as a result of our adoption of BDD principles. This was mainly because we already had Agile teams in place and everyone wanted to be involved with BDD. We found that we needed to be more focused on spreading BDD knowledge across the team as quickly as possible.

Team members would start with the simple bits, such as learning the structure of Gherkin, (which, for a developer is a relatively short task). Becoming proficient in writing examples is much harder. Focusing on the value in turning stories towards the correct focus is the thing that requires time and experience and more importantly, it's very easy to get wrong.

Introducing BDD has had a huge impact on our project delivery. The focus has very much shifted to creating business value and our teams are a lot more commercially aware. Features are very clearly understood by technical and business stakeholders alike. We now have very little conversation as a result of incorrect implementation within our projects.

Prior to moving to BDD, we were often debating internally how we classify defects (Bhalin from the testing team wrote about the differences here), i.e. whether requirements were missed etc. This has now evaporated.

Learn More

BDD training

If you would like to learn more about BDD and how it can be applied to your projects, Inviqa provides both public and custom BDD training courses for individuals and large teams.

Project work & consultancy

If you have a project that you think can benefit from BDD, we can help. Inviqa has worked alongside a number of large organisations to implement BDD as part of a project delivery.

Related reading

- http://behaviourdriven.org/
- http://dannorth.net/
- http://lizkeogh.com/
- http://theitriskmanager.wordpress.com/
- http://gojko.net/

From our blog

- Whitepaper: Building Digital Products for the Future
- Maximising the outcome of the 'three amigos' Agile workshop
- Whitepaper: Ensuring ROI in digital projects
- Closing the communication gap: an Agile approach to defining software requirements
- 5 simple tips on starting a new scrum team
- From Waterfall to Agile: a guide for project managers
- How to create user stories

Interested in helping your organisation or technical team to adopt BDD? Check out our consultancy and coaching services, and be sure to get in touchto explore how we can support you.

Glossary
If you're not an engineer or are relatively new to Agile, there may be some terms referenced that are unfamiliar. Here's a quick guide to the main terms you will come across in BDD.

Abstract Actors
Set of people grouped by their interest or usage patterns (behaviours), representing these interest or usage patterns in the software. This helps us to instead of focusing on individuals (which we could have millions in the online applications), to focus on particular group (online customers, marketing representative).

API
An API is designed to represent the foundation for building software, which would then be built on by developers. An API allows you to plug one application into another by essentially providing a pre-set integration.

Backlog
An ordered list of user-stories – read more.

Behaviours
A set of interactions that a particular person or group of people perform as part of their job function.

Cynefin
A sense-making framework introduced by Dave Snowden to describe a perspective on the evolutionary nature of complex systems, including their inherent uncertainty – read more.

Domain Driven Design
Approach to software development by connecting implementation (or code) to the evolving software model. This helps to tackle complexity in the software by simply adapting the abstractions and rules existing in the real life to the codebase being built.

Feedback Delay
Time between work is agreed upon and when it could be used to give a feedback.

Gherkin
Domain-specific language to describe business-facing behavioural examples.

Feature
An entity of functionality.

Impact Mapping
Impact mapping is a project management technique that is designed to aid the planning and delivery of projects. Impact mapping enables agile project management by creating a hierarchical project backlog. Impact mapping highlights options to enable you to create an effective project roadmap towards a specific business goal.

Neuro-Linguistic Programming
Neuro-linguistic programming is a communication technique focused on influencing or even changing neurological processes and behavioural patterns using language.

Specification
A specification is a grouped set of requirements that would define what features needed to be implemented as part of a project.

Specification by Example
Specification by Example is a method of defining business requirements and tests for software development projects. The requirements are captured by using real life examples instead of broad overviews.

Stakeholder
Person inside or outside the organisation with a direct interest in the project. Also usually a person that could have a positive or negative impact on the project.

Ubiquitous Language
Language created by both technical and business representatives on the project and inheriting attributes of both.

User-Centred Design
Approach to software and interface design based on user intentions and usage (behaviour) patterns.

User-story
A short description representing user needs based on their job function - Read more.

Waterfall
Sequential software design process with predefined stages of the delivery - Read more.

XP Method
XP (Extreme Programming) is an Agile methodology that is designed to help make quality software more adaptable to customer requirements over time. As part of XP, you would have frequent releases and short development cycles - you would also have checkpoints where the customer would be able to introduce new requirements.