THE BDD BOOKS

Discovery

Explore behaviour using examples

Gáspár Nagy
and Seb Rose

Foreword by Johanna Rothman

The BDD Books - Discovery

Explore behaviour using examples

Gáspár Nagy and Seb Rose

This book is for sale at http://leanpub.com/bddbooks-discovery

This version was published on 2018-02-23

Leanpub

This is a Leanpub book. Leanpub empowers authors and publishers with the Lean Publishing process. Lean Publishing is the act of publishing an in-progress ebook using lightweight tools and many iterations to get reader feedback, pivot until you have the right book and build traction once you do.

Tweet This Book!

Please help Gáspár Nagy and Seb Rose by spreading the word about this book on Twitter!

The suggested hashtag for this book is #bddbooks.

Find out what other people are saying about the book by clicking on this link to search for this hashtag on Twitter:

#bddbooks

Contents

Preface **1**
Who this book is for 3
Why you should listen to us 3
Acknowledgements 4
How this book series is organised 4
What is not in this book 5
Online resources 5

Chapter 2 – Structured conversation **6**
2.1 – Where is my pizza? 6
2.2 – A useful meeting 6
2.3 – Collecting examples 9
2.4 – Deliberate discovery 17
2.5 – Example Mapping in a nutshell 17
2.6 – How to establish structured conversations 20
2.7 – What we just learned 24

Preface

Behaviour Driven Development (BDD) is an agile approach to software development that insists that detailed requirements for a feature should be defined collaboratively by the business and delivery teams. The output of this collaboration is documented using business terminology that can be understood unambiguously. Finally, the documentation is structured in a way that enables it to act as automated tests that verify that the system behaves as intended. This book explores, in detail, the collaborative aspect of BDD.

BDD has been proven to be successful in thousands of projects around the world, on different platforms, in diverse industries and various project sizes. BDD is based on a set of practices that originate from the experience of many people over many years, working to *uncover better ways of delivering software*. However, there is a learning period (or more accurately, a practicing period) for BDD, so it will take some time before you start seeing a return on your investment.

We belong to the second generation of the software industry (we could call ourselves Generation Y - there are a lot of similarities). We don't believe in buzzwords or well-named methodologies, but we like to try them out to see if they work. So, if you try out BDD, how can you decide if it has worked or not?

The first indicator you are likely to notice is a reduction in ***cycle time***. The shared understanding that is gained during collaborative requirement definition sessions ensures a smooth flow from definition to delivery. If a developer or a tester discovers an ambiguity in a requirement once they have started working on it, they'll need to resolve it. This interrupts their work, as well as the work of any colleague that they ask to help. The elimination of interruptions and context switches leads to a more efficient, more predictable delivery process.

Another visible indicator is a reduction in the number of production issues. Although it is very hard to gather scientific evidence of this (because it is hard to find a "control project"), we have seen significantly fewer production issues in projects that have successfully adopted a BDD approach. For a broad review of outcomes, from a wide variety of teams, you can't do better than reading Gojko Adzic's "Specification By Example" [1].

BDD helps preserve the quality and maintainability of the software, so a further indicator is that the implementation costs of new features is kept low. This is in contrast to many other projects where, as the codebase grows, the cost of adding (or modifying) a feature increases exponentially. If allowed to deteriorate in this way, your project will finally reach the point where it is not possible to add new features anymore in a cost-efficient way and people will start talking about a "rewrite".

Our goal with this book is to ease your way through the learning period, avoiding the mistakes that we made while we were learning.

One typical mistake is to see BDD as a tool-thing. BDD is primarily about collaboration and domain discovery; any "BDD tool" can be only useful in supporting this process. You have to start by investing in collaborative discussions and the creation of a shared vocabulary. Just going after automation (using Cucumber or SpecFlow) does not work.

It doesn't.

Honest.

As we said, we don't believe in buzzwords, but if you intend to evaluate the BDD approach objectively, it is important to do it at full throttle during the evaluation period. You might feel uncomfortable or skeptical when you start doing BDD (like with any other new approach). That is absolutely fine, but don't let the evaluation be hampered by your fears. Once you have decided to evaluate how BDD could work for your team, give yourself enough time to get comfortable with the approach. Try, as far as possible, to follow our recommendations.

You're at the beginning of a brave new world. Let us help you to explore that world and discover the benefits that are waiting.

[1] Adzic, Gojko. *Specification by Example: How Successful Teams Deliver the Right Software.* Shelter Island, NY: Manning, 2011. Print.

Who this book is for

This book is written for everyone involved in the specification and delivery of software (including product owners, business analysts, developers and testers). The book starts by explaining the reasons that BDD exists in the first place and describes techniques for getting the most out of collaboration between the ***delivery team*** (those that implement the solution) and the ***business team*** (those that work on the requirements).

Just to re-iterate, this book is aimed at everyone involved in the project, irrespective of their role, whether they come from a software background or not.

It's also worth stressing that this book is tool agnostic. Whether you use Cucumber, SpecFlow, JBehave, Fit, FITNESSE, RSpec, Jasmine, Behave, or any other BDD tool – this book will help your team collaborate.

Why you should listen to us

Gáspár is the creator and main contributor of SpecFlow, the most widely used ATDD/BDD framework for .NET.

He is an independent coach, trainer and test automation expert focusing on helping teams implementing BDD and SpecFlow through his company, called Spec Solutions. He has more than 15 years of experience in enterprise software development as he worked as an architect and agile developer coach.

He shares useful BDD and test automation related tips on his blog (http://gasparnagy.com) and on Twitter (@gasparnagy). He edits a monthly newsletter (http://bddaddict.com) about interesting articles, videos and news related to BDD, SpecFlow and Cucumber.

Seb has been a consultant, coach, designer, analyst and developer for over 30 years. He has been involved in the full development lifecycle with experience that ranges from Architecture to Support, from BASIC to Ruby.

During his career, he has worked for companies large (e.g. IBM, Amazon) and small, and has extensive experience of failed projects. He's now a partner in Cucumber Limited, who help teams adopt and refine their agile practices, with a particular focus on collaboration and automated testing.

He’s a regular speaker at conferences, a contributing author to “97 Things Every Programmer Should Know” (O’Reilly) and the lead author of “The Cucumber for Java Book” (Pragmatic Programmers).

He blogs at cucumber.io and tweets as @sebrose.

Together Seb and Gáspár have over 50 years of software experience.

Acknowledgements

This book would not have been possible without the help of our reviewers:

- Gojko Adzic
- Garret Burns
- Darren Cauthon
- Lisa Crispin
- Claude Hanhart
- Dave Hanlon
- Sam Holder
- Alexandra Fung
- Adrienn Kolláth
- Gilbert Liddell
- Viktor Nemes
- Paul Rayner
- Chuck Suscheck
- Steve Tooke
- Andreas Willich

How this book series is organised

This is the first of the BDD Books series, that will guide you through the end-to-end adoption of BDD, including specific practices needed to successfully drive development using collaboratively authored specifications and living documentation.

Once you’ve implemented the approach we described you can read:

- Formulation (Book 2): express examples using Given/When/Then[2]
- Automation with SpecFlow (Book 3)[3]

What is not in this book

- Formulation
- Structuring documentation
- Gherkin
- Tools
- Automation
- Code

Online resources

http://bddbooks.com

Gáspár Nagy and Seb Rose, August 2017

[2] Nagy, Gáspár, and Seb Rose. *The BDD Books: Formulation*. In preparation. http://bddbooks.com/formulation.

[3] Nagy, Gáspár, and Seb Rose. *The BDD Books: Automation with SpecFlow*. In preparation. http://bddbooks.com/specflow.

Chapter 2 – Structured conversation

In this chapter, we are going to peer into the daily work of a software product team to learn more about how they use structured conversations to help them discover what the expected behaviour of the next feature should be. We'll start by describing one of their requirement workshops. This will introduce concepts that you're not familiar with, but don't worry, all your questions will be answered later in the chapter.

2.1 – Where is my pizza?

The team we will be visiting is developing a pizza delivery management application for a large pizza company. The application will allow clients to track the real-time location of their order(s), so they have come up with a fun name for the application: "Where is my pizza?" Some joker on the team noticed that this abbreviates to WIMP – "a weak, cowardly, or ineffectual person" (Merriam-Webster). The rest of the team still know that the product will be awesome.

There are lots of other exciting features too, but for the rest of this book we'll be considering a client's ability to modify the delivery address of an order after the order has been submitted.

2.2 – A useful meeting

It's 9 a.m. on Wednesday morning and the team is assembling in the team room for another ***requirement workshop***. There's a good turnout today - Patricia (the PO), Dave and Dishita (from Development), Tracey (from Test) and Ian (the new intern).

Requirement workshop

The team meets regularly (usually several times a week) to discuss the work that they'll be undertaking in the next sprint or two. The purpose of this meeting is to explore a story, understand it's scope and illustrate it unambiguously with concrete examples. While they're doing this, they may discover new details about the story. They may also ask questions that no one at the meeting is able to answer right away.

What matters most in this meeting is to bring diverse perspectives together, so that they can learn about what needs to be done and work together more effectively. In other organizations, similar meetings have been variously called ***three amigos meeting***, ***discovery workshop***, ***specification workshop***, ***story refinement***, ***product backlog refinement*** and ***backlog grooming*** – as always, the name is less important than the purpose.

They're very comfortable with this meeting format, because they meet several times a week for short, focused sessions that often work only on a single user story. The idea came from a blog post by Matt Wynne Introducing Example Mapping[4] that Dishita had recently read.

Patricia grabs the box of colored index cards and marker pens from the stationery cupboard and puts them in the middle of the table. Everyone knows which story Patricia has been preparing because she sent out an email yesterday. Patricia reads out the story that is going to be discussed:

> In order to fix an incorrect delivery address,
>
> As a client,
>
> I want to be able to change the delivery address after the order has been placed.

Dishita summarizes this on a yellow index card:

[4] https://cucumber.io/blog/2015/12/08/example-mapping-introduction

Figure 6 – Story card on the table

"The system will need to be able to check whether it's possible to change the delivery address", says Patricia. "We'll have to check that the new address is not too far from the current one. And we'll need to check the state of the order too." "This will be easy", says Dave and they all smile. They have heard this sentence many times before. "We'll see", answers Tracey, "let's try to come up with a few examples!" And the workshop begins.

Explaining a team discussion in a book is hard, because you need to keep track of the goals and perspectives of many people with different roles and backgrounds. To make it easier to follow, we have chosen the names of our team members so that their initials describe their role. **P**atricia is the **P**roduct Owner, **D**ave and **D**ishita are **D**evelopers, **T**racey is a **T**ester and **I**an is an **I**ntern from the University.

As we mentioned in Section 1.6, *What is BDD, then?*, one typical mistake is to look at BDD as a tool-thing. A similar mistake would be to think of BDD as a mechanical process, such as filling out a *Given*/*When*/*Then* template. We need all team members to actively challenge their understanding of the user story by coming up with concrete examples.

There are many ways you can organize your team for better collaboration. Every team and every project is different. We are going to focus on a technique called ***Example Mapping*** that is a simple and efficient way to facilitate your requirement workshops. This is one technique for carrying out a structured conversation, and it has worked very well for us, but you may need to find an alternative that is more suited to your context. Before you do, refer to Chapter 4, *Who does what and when*, where we will show how requirement workshops can fit into an agile project following Scrum, Lean/Kanban or even into a fixed scope project with distributed teams.

So let's get back to our team…

2.3 – Collecting examples

"Let's start with the happy path, where the customer should be allowed to change the delivery address… and see if it's as 'easy' as Dave thinks it will be. What would be a typical example for this?", asks Tracey.

"Yes, this is the case where the order is in preparation", starts Patricia. Tracey, who volunteered to facilitate the meeting today, takes a green card and writes *Order is in preparation* on the top of it.

"Which persona shall we use to describe this example?", asks Patricia and they all look at the wall where posters of different user types, called ***personas***[5] are displayed. The team introduced the personas a year ago when Ulla, the UX expert joined the company.

"Let's use Peter! He regularly orders pizzas at home and at the office. He probably gets the delivery address mixed up from time to time", suggests Dishita.

"OK. So let's say that Peter is in the office, but orders a Margherita pizza for home by mistake. The order has been placed and the restaurant starts to prepare it. He checks his emails a few minutes later and realizes he's used the wrong address. He clicks on the tracking link from the email and chooses 'Change Address' on the tracking page. He selects the work address and submits the changes. The change is accepted", explains Patricia.

[5] https://en.wikipedia.org/wiki/Persona_(user_experience)

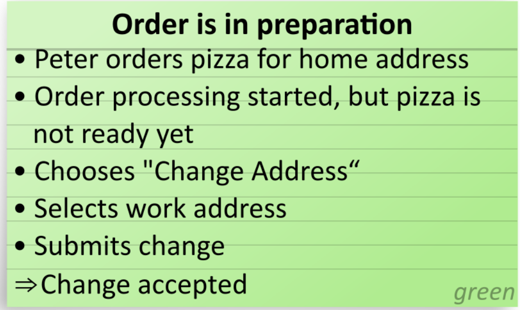
“What do you mean by ‘a few minutes later?’” asks Dishita.

“The order has just been received, so the pizza isn’t ready yet” answers Patricia.

“Ah, OK. So the pizza might be in the oven, but it’s not ready for delivery?” checks Tracey.

“Yes, that’s right” replies Patricia.

While she is explaining the details, they all look at the printed UI wireframes for the new “update address” page, to help follow the example easily. Tracey captures the important steps on a green card:



Order is in preparation

- Peter orders pizza for home address
- Order processing started, but pizza is not ready yet
- Chooses "Change Address“
- Selects work address
- Submits change

⇒Change accepted

green

Figure 7 – The first example card

They are all following Tracey as she captures these steps, so that they can verify if the details have been captured properly.

“Is it ‘not ready’ or ‘in preparation’ that we have to watch for?”, asks Ian. “We use one in the example title and the other in the second step of the example”

“Those two states mean essentially the same thing”, answers Dave. “If the pizza is in preparation, then it’s obviously not ready.”

“We just learnt about state diagrams in our UML module at university. Is that something we could use here?” asks Ian.

Dishita grabs a pen and stands at the whiteboard. “We don’t need a full UML state diagram, but I think an overview of the states would be useful, as well as some of the events.” It takes a few minutes to come up with a state diagram, similar to the one below.

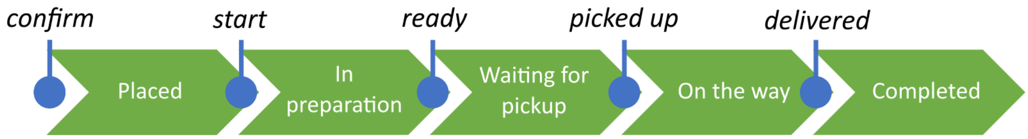
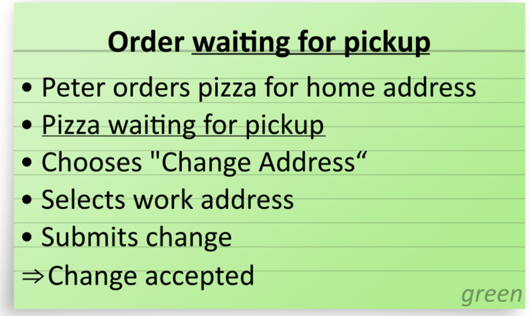


Figure 8 – State diagram of pizza process

“Then why don’t we let them change the address up until the delivery person picks up the pizza?” asks Tracey. They all look at Patricia.

“That’s a good point, Tracey. Actually the important turning point is when the delivery person picks up the order. This is when they check the delivery address and plan the route”, concludes Patricia.

“OK. I’ll fix the example card”, says Tracey and changes the card to look like this:



Order waiting for pickup

- Peter orders pizza for home address
- Pizza waiting for pickup
- Chooses "Change Address“
- Selects work address
- Submits change

⇒Change accepted

green

Figure 9 – The corrected example card (changes underlined)

“Is everybody happy with this?” Everyone nods, so she places the card below the story card on the desk. “What rule is this example illustrating?”

“‘Allow address change if not picked up yet’”, replies Patricia and she picks a blue card, writes this rule on it and places it above the green card.

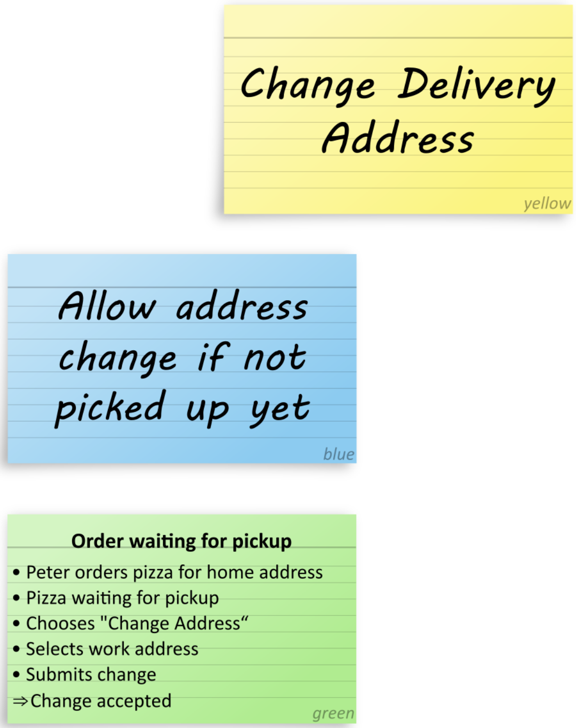

Change Delivery Address
yellow
Allow address change if not picked up yet
blue
Order waiting for pickup
• Peter orders pizza for home address
• Pizza waiting for pickup
• Chooses "Change Address“
• Selects work address
• Submits change
⇒Change accepted
green

Figure 10 – First rule in the example map

“What kind of other examples can we imagine for this rule? Is there a counter-example?”, Tracey helps to move the meeting forward.

“Sure! If the order has been already picked up, we should respond with a big fat error message”, Dave replies promptly and smiles. “Easy…”

Everyone tries to imagine the situation… somehow it feels wrong. Dishita finally comes up with an example.

“Let’s look at our other persona, Tim. He is a first time buyer, so he has to type in the delivery address. What if he makes a small mistake? I once mixed up the house number in one of my orders and only realized it when I got hungry and checked the notification mail.”

“But we can’t just let them change the address to a completely different location, once the pizza is already at the doorstep of the original delivery address…” says Patricia.

They start a discussion about possible options, but the discussion gets stuck. “There are lots of ways we could handle a late change of delivery address, but none of them are simple. Do we really need to do this now?” asks Dave.

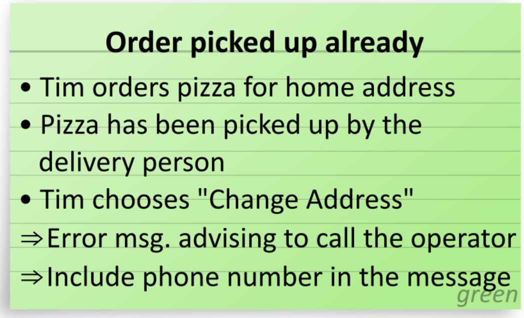
“It looks like we can’t solve this now. Let’s make a red card for it, and check the statistics to see how often this happens and how much extra cost is caused by this sort of mistake”, Tracey suggests.

They all agree, so she takes a red card and summarizes the problem. They use the red cards to track questions or other discussion points that they cannot solve immediately. They place the red cards on the desk, so that everyone can see them. This way they can avoid endless discussions about the topic.

“Can we come up with a temporary workaround for the problem?”, asks Tracey once she has finished writing down the question.

“Maybe we could change the error message to advise the users to call the operator”, suggests Ian.

“Very good! Let’s capture that quickly before we forget!” replies Patricia and they capture another example and place it under the previous one.



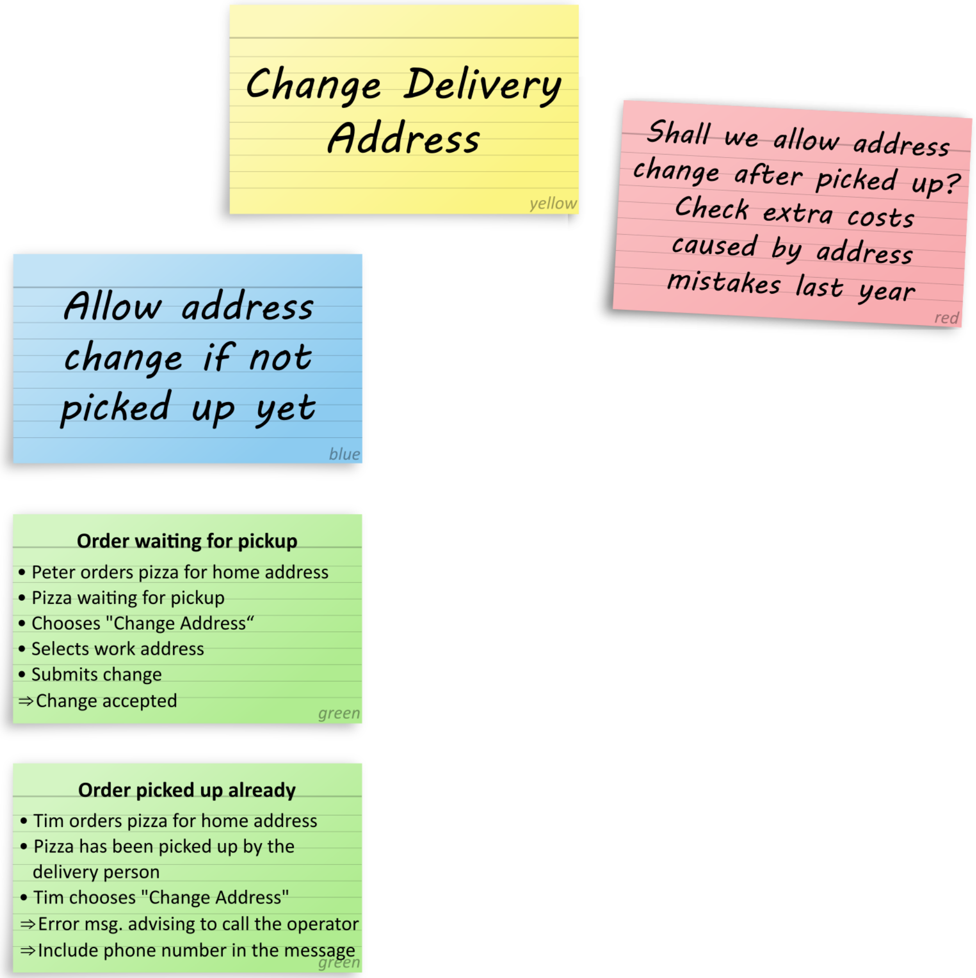
Order picked up already

- Tim orders pizza for home address
- Pizza has been picked up by the delivery person
- Tim chooses "Change Address"

⇒Error msg. advising to call the operator

⇒Include phone number in the message

Figure 11 – The second example card



Change Delivery Address

Shall we allow address change after picked up? Check extra costs caused by address mistakes last year

Allow address change if not picked up yet

Order waiting for pickup

- Peter orders pizza for home address
- Pizza waiting for pickup
- Chooses "Change Address“
- Selects work address
- Submits change

⇒Change accepted

Order picked up already

- Tim orders pizza for home address
- Pizza has been picked up by the delivery person
- Tim chooses "Change Address"

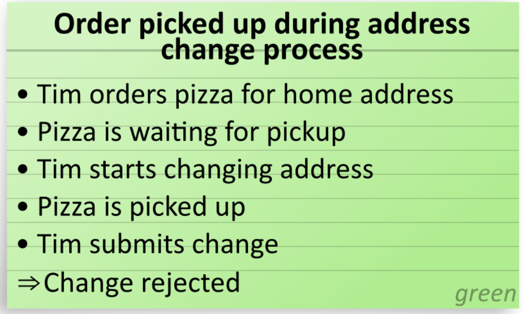
⇒Error msg. advising to call the operator
⇒Include phone number in the message

Figure 12 – A counter example and a question

They all look at the examples to see how the system will behave in the different situations.

“There’s another one!” cries out Tracey. “What if the pizza was picked up while Tim was typing in the new address in the address change screen?”

The team realize that this illustrates the need for state verification both when the address change screen is opened and also when the change is submitted. This means that even though it's an edge case, it's important enough to capture it as a third example of the rule.



Order picked up during address change process

- Tim orders pizza for home address
- Pizza is waiting for pickup
- Tim starts changing address
- Pizza is picked up
- Tim submits change

⇒Change rejected

green

Figure 13 – The third example card

They cannot come up with any further relevant example, so they move on.

"Is this what you call 'easy', Dave?", asks Patricia.

"Well… at least we've learned a lot about the address change process", acknowledges Dave.

The team then discuss other business rules, like *only valid address is accepted*, *estimated time of arrival is updated* or *new address should be within restaurant's delivery range*. They come up with examples for all these rules and lay them on the desk.

The workshop finishes in about half an hour, Tracey takes a photograph of the example map (Figure 14) and everyone goes back to their desk.

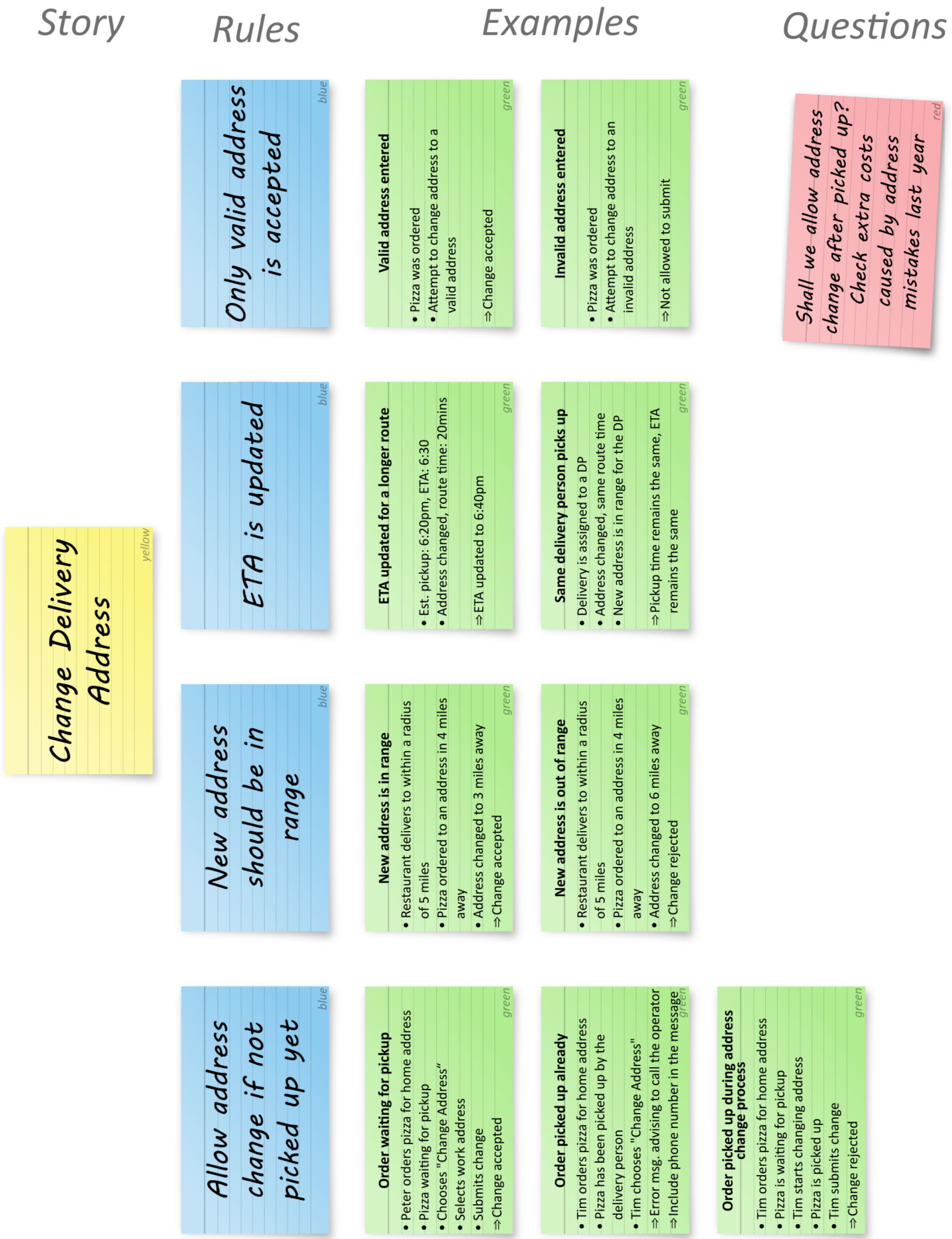

Story
Rules
Examples
Questions
Only valid address is accepted
blue
Valid address entered
• Pizza was ordered
• Attempt to change address to a valid address
⇒Change accepted
green
Invalid address entered
• Pizza was ordered
• Attempt to change address to an invalid address
⇒Not allowed to submit
green
Shall we allow address change after picked up? Check extra costs caused by address mistakes last year
red
ETA is updated
blue
ETA updated for a longer route
• Est. pickup: 6:20pm, ETA: 6:30
• Address changed, route time: 20mins
⇒ETA updated to 6:40pm
green
Same delivery person picks up
• Delivery is assigned to a DP
• Address changed, same route time
• New address is in range for the DP
⇒Pickup time remains the same, ETA remains the same
green
Change Delivery Address
yellow
New address should be in range
blue
New address is in range
• Restaurant delivers to within a radius of 5 miles
• Pizza ordered to an address in 4 miles away
• Address changed to 3 miles away
⇒Change accepted
green
New address is out of range
• Restaurant delivers to within a radius of 5 miles
• Pizza ordered to an address in 4 miles away
• Address changed to 6 miles away
⇒Change rejected
green
Allow address change if not picked up yet
blue
Order waiting for pickup
• Peter orders pizza for home address
• Pizza waiting for pickup
• Chooses "Change Address“
• Selects work address
• Submits change
⇒Change accepted
green
Order picked up already
• Tim orders pizza for home address
• Pizza has been picked up by the delivery person
• Tim chooses "Change Address"
⇒Error msg. advising to call the operator
⇒Include phone number in the message
green
Order picked up during address change process
• Tim orders pizza for home address
• Pizza is waiting for pickup
• Tim starts changing address
• Pizza is picked up
• Tim submits change
⇒Change rejected
green

Figure 14 – The final example map

2.4 – Deliberate discovery

Developing software is a process of learning. I've never met a team that, after they deliver some working software, say "if we were to do that again, we'd do it exactly the same way." That's because, in the course of developing the software, they discover things that they didn't know at the beginning.

Discovery that happens while you're developing software can be thought of as "accidental discovery" - it may upset your schedule, or even derail or interrupt your roadmap entirely. The discovery that happens during a requirement workshop is "***deliberate discovery***." The meeting is convened with the almost-certain knowledge that there are things that we don't know. We deliberately explore our understanding of the requirements using concrete examples and, more often than not, we are rewarded by learning things. The alternative is that the learning happens, accidentally, after the delivery team has already started developing the solution.

While our team were discussing the "allow delivery address change" requirement above, examples turned out to be very useful. The rule started off looking really simple, but when they started coming up with examples to illustrate the rule, they found that they didn't have a shared understanding of the possible states of an order. They cleared this up with a state diagram, which ultimately led to them modifying the rule to "Allow address change when order not picked up."

The next example dug into what should happen when the order has been picked up, which has implications for the user experience. They came up with a simple solution and wrote out a question card to be answered by the business. This naturally led them to think about what should happen if the order is picked up *while* the address change is being requested, and they discovered another important behaviour.

Examples are used both to illustrate what we already know and to force us to question our assumptions. They underpin the deliberate discovery conversation.

2.5 – Example Mapping in a nutshell

> "Example Mapping is a simple, low-tech method for making conversations short and powerfully productive." - Matt Wynne

The ***Example Mapping*** technique[6] was discovered/invented by Matt Wynne who regularly facilitates requirement workshops for his customers. At one of his workshops, he had a pack of 4-colored index cards with him. He used the green cards to capture examples and grouped them by rules, which he wrote on blue cards. He discovered that arranging these cards as a map helps guide the discussion and gives a good visual overview of the requirements.

When participating in an Example Mapping workshop, we capture different artifacts on differently colored index cards or post-it notes.

- ***Examples*** are captured on **green cards** – illustrate concrete behaviour of the system.
- ***Rules*** are written on **blue cards** – these are logical groupings of the examples usually focusing on a particular condition. Many teams call these ***acceptance criteria*** (AC), ***business rules***, or simply ***requirements***.
- ***Questions*** or ***assumptions*** are captured on **red cards** – any topic that would block the discussion. Since these are visible to everyone, we can avoid re-discussing these (usually frustrating) topics again and again.
- ***User stories*** are captured on **yellow cards** – we usually start discussing a single user story, but as we are digging into the details, we often decide to split the story into smaller stories and postpone some of them.

Seb’s story: Business rules, requirements, acceptance criteria

When my teams start discussing a user story what they typically want to understand is the scope of the story and how difficult it might be to implement. Examples really help us understand how the story is supposed to behave, but we want them in related groups.

These groupings have been called different things by different teams: business rules, acceptance criteria, requirements. Each of the terms comes with its own baggage and often hampers communication on the team. Instead, we’ve started just calling them *rules* - they’re abstract statements that describe a single aspect

[6] https://cucumber.io/blog/2015/12/08/example-mapping-introduction

of behaviour.

An added bonus is that when it comes to splitting a story, we can often split them simply by moving some rules (and their associated examples) to a new story.

We place the story card on the top row and arrange the rules in a row underneath. The examples belonging to a particular rule are placed below the rule card they relate to. We put the red cards to the side of the example map. At the end of the discussion the desk should look like Figure 14.

Gaspar’s story: Rejoining a discussion

We all have been in meetings where a notification on our phone or some other interruption distracts our focus. Once you fall out of an animated discussion, it is very hard to get back into it. These kinds of problems can be minimized by establishing a better meeting culture, but they cannot be avoided completely. So, we’d better accept that this might happen and create an infrastructure that helps the people get back into the discussion as quickly as possible. Having a visual map (or a mindmap) that is visible to everyone makes it easier to see the big picture and check the details at the same time.

The example cards will be used later when we come to write our scenarios. It does not matter what format you use to write examples as long as you capture all the details that seemed important in the discussion. For example, even if you use Cucumber, you should not use *Given*/*When*/*Then* to write your examples.

When we have captured an example, we give it a title, so that we can refer to it easily. One simple way of coming up with a title is to copy the way that episodes of the *Friends* sitcom were named: *“The One Where Rachel Finds Out”*. The words you put after the *“The One Where …”* happen to be very good titles for our examples. *“The One Where the* ***Order has been Picked Up Already***“.

There’s no strict order you should collect the examples and the rules in. If the story is simple or well prepared, it will probably arrive with an initial set of rules (or acceptance criteria). In this case it makes sense to go through these rules and understand their details by creating examples that illustrate each of them.

In other situations, where the story is more vague, it might make sense to come up with some examples that describe the typical behaviours that will be expected. Then, identify the rules that govern them.

No matter how you do it, it is practical to nominate a ***facilitator***, who keeps the meeting going. The facilitator takes care that the discussion is captured on the cards and that everyone agrees with the form they have been written down. The facilitator is not a special role, anyone from the team can do it. We recommend you rotate this role across all team members.

With Example Mapping, you can discuss detailed requirements in a surprisingly short time. In many cases, the details of a user story can be discussed in 20-30 minutes. We've found this style of workshop can work for teams no matter what flavor of agile they're using, as we discuss in Chapter 4, *Who does what and when*. Because the workshops are short, we can run them several times a week.

Once the map has been created, it is important not to lose this information. Some teams take photos and share it with team members. Others pin the cards on a pinboard in the team room or save it as a mind map.

2.6 – How to establish structured conversations

Regardless of whether you use Example Mapping or another technique, structuring the conversations will help your requirement workshops be more focused and efficient. It's time to define exactly what we mean by ***structured conversation***. A structured conversation is a facilitated exchange of ideas that conforms to a predefined form. In the context of a requirement workshop, a structured conversation exhibit the following properties that we expand on below:

- **collaborative**: all attendees participate actively
- **diverse perspectives**: we need the three amigos

- **short**: we want regular workshops, so that the learning feedback loop stays fast
- **progressive focus**: we capture the progress of the workshop in real time, allowing the discussion to move forward quickly
- **consensus**: agreed concrete examples are the measure of the workshop's success

Collaborative

The conversations should include the entire team and encourage them to collaborate actively. In many agile projects the term *collaboration* means no more than inviting everyone to a planning meeting at which the product owner explains the requirements and everyone else listens. We need more bi-directional communication.

Gaspar's story: Help them participate in the discussion!

Once a member of my team asked me an important question after the meeting finished. "This is a pretty important question! Why didn't you ask the PO during the meeting?" I asked back trying to hide my emotions. "Because I wasn't sure I understood everything and I thought my question might have been answered anyway" he responded. I was pretty much frustrated about this and other similar situations and I was dreaming of a team consisting of communicative, active super-heroes asking questions frequently.

Soon afterwards, I was invited to a non-IT discussion once, where neither the topic nor the language were well known to me. I feared that I would not be able to participate properly in the discussion. The organizer helped me out by giving me a handout that contained the agenda, the discussion points and a few important sentences. Without these, I might have looked dumb, but instead I was able to participate in the discussion. Impressed by this experience I revised my attitude to team work.

Identifying the root causes of communication problems and figuring out how to

> solve them is far better than dreaming of perfection and whining when you can't achieve it. Working with an all-super-hero team must be really boring.

Everyone has a slightly different personality. You should establish a culture where everyone can participate, regardless of their personality. By using a method that requires people to stand up and move, talk to each other and arrange/annotate colored cards, we can exercise different senses, so everyone can contribute in the areas that they feel most comfortable with. For distributed teams, the possibilities are more limited, but with a few smart ideas the same efficiency can be achieved (see more on this in Chapter 4, *Who does what and when*).

Diverse perspectives

We uncover ambiguities by organizing requirement workshops where the representatives of diverse perspectives (business, test, development) come together. These meetings are sometimes called ***Three Amigos***[7] meetings. Despite the name, the point is not that there should be only three people in the room. In your team there might be other roles as well (e.g. UX) or you might invite multiple representatives of the same role. What makes these meetings powerful is that the representatives of the different roles can challenge their understanding of the requirements at an early stage. Even though they are all talking about the same requirement, each person has their own perspective:

- the business representative focuses on the fulfillment of the business goals – for example when Patricia agrees that the address can be changed before the order is picked up
- the developers explore the technical implications of the feature – for example when Dave concludes that all solutions for late change of delivery address will introduce costly complexities
- the testers can challenge the feasibility of testing the feature and help identify special corner cases – for example when Tracey noticed that the order might be picked up while the client was changing the delivery address

[7] https://www.infoq.com/interviews/george-dinwiddie-three-amigos

Short

We suggest that requirement workshops should be no longer than 30 minutes. This is a because:

- long meeting are exhausting - you'll often not get active participation from all attendees throughout the meeting
- long meetings are expensive – multiply the time by the number of people in the room and you frequently find meetings that cost a person-day
- short meetings can be scheduled more frequently – shorter meetings are less disruptive and are easier to schedule
- frequent meetings can vary the attendees – we can improve shared ownership by ensuring that it's not always the same people at the meeting
- frequent meetings reduce the impact of unanswerable questions – we can continue the discussion as soon as we discover the answer

Your situation may require longer, infrequent meetings, but even then we urge you to structure each meeting into several short, focused sessions.

Progressive focus

We start the workshop knowing what we're going to discuss, but we never know quite what we'll discover. However, as the workshop unfolds we must capture what we learn, so that at the end we have a full record of our knowledge. Our understanding becomes progressively more complete, while the record allows us to maintain focus on what we still don't understand.

Seb's story: Progressive JPEG

I think of a structured conversation as being a bit like a JPEG image downloading over a slow connection. The image becomes progressively more and more detailed. Early on you can often guess what it's an image of, but it's not until it finishes downloading that you can see the picture in its full glory.

For our conversation to keep focused while our understanding develops progressively, we need to:

- know what we're meeting about – Patricia emailed out the story for discussion the day before
- capture our understanding as it develops – the format is not important, but it must be low friction and captured continuously
- be able to quickly grasp the state of the discussion – by capturing our understanding in a form that is readable by everyone, such as an example map
- stop discussions that aren't going anywhere – for example, by capturing unanswerable questions rather than discussing them fruitlessly

Consensus

We know we have achieved consensus when we agree:

- that the output is correct – the output may not be complete, but everything in it must be agreed by everybody
- whether the feature is sufficiently understood – if it is then it needs no further discussion and development can proceed
- there is no hidden/private knowledge – the content of all conversations is captured in the output and there is no need to consolidate private notes, email chains etc.
- who is responsible for answering each remaining question – capturing questions is a great way to keep the discussion moving, but someone needs to ensure that they get answered

2.7 – What we just learned

Software development is a learning process. The more we can learn about the problem, the easier solving it becomes. This process can be made more effective by having several team members (with different perspectives) analyzing the requirements together *before they start developing the software*. These collaborative requirement workshops are most productive if they are kept short and run regularly throughout the project – often several times a week.

There are many structured conversation techniques that have been used to facilitate these requirement workshops, but the most useful that we have come across is Example Mapping, as described by Matt Wynne. It's simple, well documented and results in a visual recording of the conversation that clearly communicates the agreements that were reached, as well as the questions that remain unanswered. This in turn makes it easier for those that weren't at the meeting to understand the issues and provide useful feedback.

All software development projects depend on well understood requirements, but the approach described here works particularly well for the lightweight agile and lean methods that are in common use today.