Gojko Adzic

Specification by Example

How successful teams deliver the *right* software

MANNING

Table of Contents

Preface. Specification by Example: How successful teams deliver the right software, ISBN: 9781617290084
Prepared for yeongsheng.tan@baml.com, YEONG TAN

Preface

The book you hold in your hands, or see on your screen, is the result of a series of studies of how teams all over the world specify, develop, and deliver the right software, without defects, in very short cycles. It presents the collective knowledge of about 50 projects, ranging from public websites to internal back-office systems. These projects involved diverse teams, from small ones working in the same office to groups spread across different continents, working in a range of processes including Extreme Programming (XP), Scrum, Kanban, and similar methods (often bundled together under the names *agile* and *lean*). They have one thing in common—they all got the practices of collaborating on specifications and tests right, and they got big benefits out of that.

Specification by Example

Different teams use different names for their ways of dealing with specifications and tests, yet they all share a common set of core principles and ideas, which I hold to be essentially the same thing. Some of the names that the teams used for these practices are

- Agile acceptance testing
- Acceptance Test-Driven Development
- Example-Driven Development
- Story testing
- Behavior-Driven Development
- Specification by Example

The fact that the same practices have so many names reflects the huge amount of innovation in this field at the moment. It also reflects the fact that the practices described in this book impact the ways teams approach specifications, development, and testing. To be consistent, I had to choose one name. I settled on *Specification by Example*, and I'll use that in the rest of the book. I explain this choice in detail in the "A few words on the terminology" section later in this introduction.

In the real world

I present this topic through case studies and interviews. I chose this approach so that you can see that there are real teams out there right now doing this and reaping big benefits. Specification by Example is not a dark art although some popular media might make you think that.

Almost everything in this book is from the real world, real teams, and real experiences. A small number of practices are presented as suggestions without being backed by a case study. These are ideas that I think will be important for the future, and they're clearly introduced as such.

I'm certain that the studies I conducted leading to this book and my conclusions will be dismissed for not being a serious scientific research by those skeptics who claim that agile development doesn't work and that the industry should go back to "real software engineering."[1] That's fine. The resources available to me for this book project are minute compared to what would be required for a serious scientific research. Even with those resources, I'm not a scientist, nor do I intend to present myself as such. I'm a practitioner.

Who should read this book?

If you're a practitioner, like me, and your bread and butter come from making or helping software go live, this book has a lot to offer. I primarily wrote this book for teams who have tried to implement an agile process and ran into problems that manifest themselves as poor quality, rework, and missed customer expectations. (Yes, these are problems, and plainly iterating is a workaround and not a solution.) Specification by Example, agile acceptance testing, Behavior-Driven Development, and all the alternative names for the same thing solve these problems. This book will help you get started with those practices and learn how to contribute better to your team, regardless of whether you qualify yourself as a tester, developer, analyst, or product owner.

A few years ago, most people I met at conferences hadn't heard of these ideas. Most people I meet now are somewhat aware of these practices, but many failed to implement them properly. There's very little literature on problems that teams face while implementing agile development in general, so every discouraged team thinks that they're unique and that somehow the ideas don't work in their "real world." They seem surprised how I can guess three or four of their biggest problems after just five minutes of listening to them. They are often completely astonished that many other teams have the same issues.

If you work in such a team, the first thing that this book will do for you is show you that you're not alone. The teams I interviewed for this book aren't perfect—they had tons of issues as well. Instead of quitting after they hit a brick wall, they decided to drive around it or tear it down. Knowing this is often encouraging enough for people to

[1] For more on the delusion that engineering rigor would help software development, as if it were some kind of second-rate branch of physics, see also http://www.semat.org. For a good counterargument, see Glenn Vanderburg's presentation "Software Engineering Doesn't Work!" at http://confreaks.net/videos/282-lsrc2010-real-software-engineering.

look at their problems in a different light. I hope that after reading the book you'll feel the same.

If you're in the process of implementing Specification by Example, this book will provide useful advice on how to get past your current problems and learn what you can expect in the future. I hope you will learn from the mistakes of others and avoid hitting some problems at all.

This book is also written for experienced practitioners, people with a relatively successful implementation of Specification by Example in their process. I started conducting the interviews expecting that I knew most of what's out there, looking for external confirmation. I ended it surprised by how many different ideas people implemented in their contexts, things I never thought about. I learned a lot from these examples, and hope you will too. The practices and the ideas described here should inspire you to try alternative solutions to your problems or realize how you can improve the process of your team once you read similar stories.

What's inside?

In part 1, I introduce Specification by Example. Instead of convincing you why you should follow the principles outlined in the book, I show you—in the true Specification by Example style—examples of benefits that teams got from this process. If you're thinking about buying this book, skim over chapter 1 and see if any of the benefits presented there would apply to your project. In chapter 2, I introduce the key process patterns and key artifacts of Specification by Example. In chapter 3, I explain the idea of living documentation in more detail. In chapter 4, I present the most common starting points for initiating the changes to process and team culture and advise what to watch out for when you start implementing the process.

One of my goals with this book is to create a consistent language for patterns, ideas, and artifacts that teams use to implement Specification by Example. The community has a dozen names for the practice as a whole and twice as many for various elements of it. Different people call the same thing feature files, story tests, BDD files, acceptance tests, and so on. For that reason, I also introduce what I think are very good names for all the key elements in chapter 2. Even if you're an experienced practitioner, I suggest you read this chapter to make sure that we have the same understanding of the key names, phrases, and patterns in this book.

In part 2, I present the key practices that the teams from the case studies used to implement the principles of Specification by Example. Teams in different contexts do very different things, sometimes even opposing or conflicting, to get to the same effect. In addition to the practices, I document the contexts in which the teams use them to implement the underlying principles. The seven chapters in part 2 are roughly broken down by process areas.

There are no best practices in software, but there are definitely good ideas that we can try to apply in different contexts. You will find thumbs-up and thumbs-down icons next to the sections in part 2, indicating practices that several teams from the survey found useful or issues they commonly faced. Treat these as suggestions to try out or avoid, not as prescriptions for something that you must follow. Arrow icons point to particularly important ideas for each practice.

Software development isn't static—teams and environments change and the process must follow. I present case studies showing the journeys of a few selected teams in part 3. I write about their processes, constraints, and contexts, analyzing how the processes evolved. These stories will help you get started with your journey or take the next step, find ideas, and discover new ways of doing things.

In the final chapter of the book, I summarize the key things I've learned from the case studies leading to this book.

Beyond the basics

On the traditional *Shu-ha-ri*[2] learning model, this book is at the *Ha* level. *Ha* is about breaking the old rules and showing that there are many successful models. In my book *Bridging the Communication Gap*, I presented my model and my experience. In this book, I try hard not to be influenced by my background. I present things from the projects I worked on only when there's an important point to make and I don't think any of the teams featured in the book had a similar situation. In that sense, Specification by Example continues where *Bridging the Communication Gap* stopped.

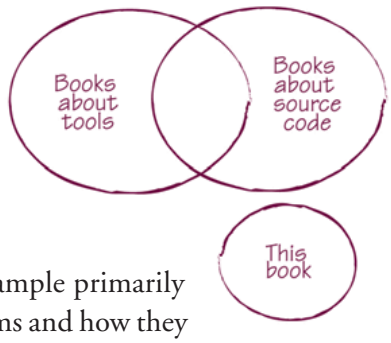
I introduce the basic principles briefly in chapter 2. Even if you've never heard of any of these ideas before, this should give you enough information to understand the rest of the book, but I won't go into the basics too much. I wrote about the basics of Specification by Example at length in *Bridging the Communication Gap* and have no wish to repeat myself.

If you want to go over the basics in more detail, visit http://specificationbyexample.com, register a copy of this book, and you'll get the PDF of *Bridging the Communication Gap* free.

I don't think that I'll write a follow-up on this subject on the *Ri* level—because that level is beyond books. On the other hand, I believe that this book will help you move to that level. Once you start thinking that the choice of a particular tool is irrelevant, you are there.

[2] Shu-ha-ri is a learning model associated with Aikido. It roughly translates to "obey-detach-leave." At the first level (Shu - "obey"), a student learns by closely following one model. At the second level (Ha - "detach"), the student learns that there are multiple models and solutions. At the third level (Ri - "leave"), the student goes beyond following models.

This book has no source code and doesn't explain any tools



This book has no source code or instructions on how to work with a particular tool. I feel compelled to mention this upfront, because I had to explain it already several times during the publishing process (typically as an answer to the question, "What do you mean? A software development book without source code? How's that possible?").

The principles and practices of Specification by Example primarily affect how people communicate in software delivery teams and how they collaborate with business users and stakeholders. I'm sure that many tool vendors will try to sell you a technical solution for that. There are also many managers who would be happy to pay for their problem to go away instantly. Unfortunately for them, this is mostly a people problem, not a technical one.

Bill Gates said, "The first rule of any technology used in a business is that automation applied to an efficient operation will magnify the efficiency. The second is that automation applied to an inefficient operation will magnify the inefficiency." Many teams who failed with Specification by Example have magnified their process inefficiency by automating it. Instead of focusing on a particular tool, I want to address the real reasons why teams struggle to implement these ideas. Once you get the communication and collaboration right, you'll be able to choose the right tool to fit it. If you want to know more about tools that support specification by example after reading this book, go to http://specificationbyexample.com and check out the resources section.

A few words on the terminology

If this is your first contact with Specification by Example, Acceptance Test-Driven Development, agile acceptance testing, Behavior-Driven Development, or any of the other names people use for this set of practices, you've avoided years of confusion caused by misleading names. You should feel good about that, and you may skip this part of the introduction. If you have already come into contact with any of those ideas, the names I use in this book might surprise you. Read on to understand why I use those names and why you should start using them as well.

While writing this book, I had the same problem practitioners often have when writing our automated specifications. The terminology has to be consistent to make sense, but we don't necessarily see that until we write things down. Because this book is the result of a series of interviews, and many people I spoke to used different names for the same thing, it was quite hard to make the story consistent with all the different names.

I realized that the practitioners of Specification by Example, myself included, have traditionally been guilty of using technical terms to confuse both ourselves and everyone else who tries to implement these practices. Then I decided that one of my goals with this book would be to change the terminology in the community. If we want to get business users more involved, which is one of the key goals of these practices, we have to use the right names for the right things and stop confusing people.

This lesson is obvious when we write our specifications, and we know that we need to keep the naming consistent and avoid misleading terms. But we don't do this when we talk about the process. For example, when we say *continuous integration* in the context of Specification by Example, we don't really mean running integration tests. So why use that term and then have to explain how acceptance tests are different from integration tests? Until I started using *specification workshop* as the name for a collaborative meeting about acceptance tests, it was difficult to convince business users to participate. But a simple change in naming made the problem go away. By using better names, we can avoid many completely meaningless discussions and get people started on the right path straightaway.

Why Specification by Example?

I first want to explain why I chose Specification by Example as the overall name for the whole set of practices, as opposed to agile acceptance testing, Behavior-Driven Development, or Acceptance Test-Driven Development.

During the Domain Driven Design eXchange 2010 conference[3] in London, Eric Evans argued that *agile* as a term has lost all meaning because anything can be called agile now. Unfortunately, he's right. I've seen too many teams who tried to implement a process that was obviously broken but slapped the name *agile* on it as if that would magically make it better. This is in spite of a huge body of available literature on how to properly implement XP, Scrum, and other less-popular agile processes.

To get around this meaningless ambiguity and arguing whether agile works or not (and what it is), I avoid using the term *agile* in this book as much as I can. I use it only when referring to teams that started implementing well-defined processes built on the principles outlined in the Agile Manifesto. So without being able to mention agile in every second sentence, agile acceptance testing as a name is out of the question.

The practices described here don't form a fully fledged software development methodology. They supplement other methodologies—both iteration and flow based—to provide rigor in specifications and testing, enhance communication between various stakeholders and members of software development teams, reduce unnecessary rework, and facilitate change. So I don't want to use any of the "Driven Development" names. Especially not Behavior-Driven Development (BDD). Don't take this as a sign that I

[3] http://skillsmatter.com/event/design-architecture/ddd-exchange-2010

have anything against BDD. Quite the contrary, I love BDD and consider most of what this book is about actually a central part of BDD. But BDD suffers from the naming problem as well.

What BDD actually means changes all the time. Dan North, the central authority on what BDD is and what it is not, said that BDD is a *methodology* at the Agile Specifications, BDD, and Testing Exchange 2009.[4] (Actually he called it "a second-generation, outside-in, pull-based, multiple-stakeholder, multiple-scale, high-automation, agile methodology.") To avoid any confusion and ambiguity between what North calls BDD and what I consider BDD, I don't want to use that name. This book is about a precise set of practices, which you can use within a range of methodologies, BDD included (if you accept that BDD is a methodology).

I also want to avoid using the word *test* too much. Many managers and business users unfortunately consider testing as a technical supplementary activity, not something that they want to get involved in. After all, they have dedicated testers to handle that. Specification by Example requires an active participation of stakeholders and delivery team members, including developers, testers, and analysts. Without putting *tests* in the title, story testing, agile acceptance testing, and similar names are out.

This leaves Specification by Example as the most meaningful name with the least amount of negative baggage.

Process patterns

Specification by Example consists of several process patterns, elements of the wider software development life cycle. The names I use for process patterns in this book are a result of several discussions at the UK Agile Testing user group meetings, Agile Alliance Functional Testing Tools mailing list, and workshops. Some of them have been in use for a while; some of them will be new to most readers.

A popular approach in the community is to use the name of a practice or tool to describe a part of the process. Feature Injection is a good example—it's a popular name for extracting the scope of a project from the business goals. But Feature Injection is just one technique to do that, and there are alternative ways to achieve the same goal. In order to talk about what different teams do in different contexts, we need a higher-level concept that includes all those practices. A good name describes the expected outcome and clearly points to the key differentiating element of this set of practices.

In the case of Feature Injection and similar practices, the outcome is a scope for a project or a milestone. The key differentiator from the other ways of defining the scope is that we focus on the business goals. So I propose that we talk about *deriving scope from goals*.

[4] http://skillsmatter.com/podcast/java-jee/how-to-sell-bdd-to-the-business

One of the biggest issues teams have with Specification by Example is who should write what and when. So we need a good name that clearly says that everyone should be involved (and that this needs to happen before the team starts programming or testing), because we want to use acceptance tests as a target for development. *Test first* is a good technical name for it, but business users don't get it, and it doesn't imply collaboration. I propose we talk about *specifying collaboratively* instead of test first or writing acceptancIt sounds quite normal to put every single numerical possibility into an automated functional test. Why wouldn't we do it if it's automated? But such complex tests are unusable as a communication tool, and in Specification by Example we need to use tests for communication. So instead of writing functional tests, let's talk about *illustrating using examples* and expect the output of that to be *key examples* to point out that we want only enough to explain the context properly.[5]

Key examples are raw material, but if we just talk about acceptance testing then why not just dump complicated 50-column, 100-row tables with examples into an acceptance test without any explanation? It's going to be tested by a machine anyway. With Specification by Example, the tests are for humans as well as for machines. We need to make it clear that there's a step after illustrating using examples, where we extract the minimal set of attributes and examples to specify a business rule, add a title and description, and so on. I propose we call this step *refining the specification*.[6]

The result of this refinement is at the same time a specification, a target for development, an objective way to check acceptance, and a functional regression test for later. I don't want to call this an acceptance test because it makes it difficult to justify why this document needs to stay in domain language, be readable, and be easily accessible. I propose we call the result of refining a *specification with examples*, which immediately points to the fact that it needs to be based on examples but also contain more than just raw data. Calling this artifact a specification makes it obvious that everyone should care about it and that it needs to be easy to understand. Apart from that, there's a completely different argument as to whether these checks are there to automatically accept software or to automatically reject the code that doesn't satisfy what we need.[7]

I just don't want to spend any more time arguing with people who've already paid a license for QTP that it's completely unusable for acceptance tests. As long as we talk about test automation, there's always going to be a push to use whatever horrible contraption testers already use for automation, because it's logical to managers that their teams use a single tool for test automation. Agile acceptance testing and BDD tools don't compete with QTP or tools like that; they address a completely different problem.

[5] Thanks to David Evans who suggested this.
[6] Thanks to Elisabeth Hendrickson who suggested this name.
[7] http://www.developsense.com/blog/2010/08/acceptance-tests-lets-change-the-title-too

A specification shouldn't be translated into something technical just for automation. Instead of talking about test automation, let's call automating a check without distorting any information *automating validation without changing specifications*. The fact that we need to automate validation without changing the original specification should help us avoid the horror of scripting and using technical libraries directly in test specifications. An executable specification should be unchanged from what it looked like on the whiteboard; it shouldn't be translated into Selenium commands.

After the validation of a specification is automated, we can use it to validate the system. In effect, we get *executable specifications*.

We want to check all the specifications frequently to make sure that the system still does what it's supposed to do and, equally important, to check that the specifications still describe what the system does. If we call this regression testing, it's very hard to explain to testers why they shouldn't go and add five million other test cases to a previously nice, small, focused specification. If we talk about continuous integration, then we get into the trouble of explaining why these tests shouldn't always be run end to end and check the whole system. For some legacy systems, we need to run acceptance tests against a live, deployed environment. Technical integration tests run before deployment. So let's not talk about regression testing or continuous integration; let's talk about *validating frequently*.

The long-term payoff from Specification by Example comes from having a reference on what the system does that's as relevant as the code itself but much easier to read. That makes development much more efficient long term, facilitates collaboration with business users, leads to an alignment of software design and business models, and just makes everyone's work much easier. But to do this, the reference really has to be relevant, it has to be maintained, and it has to be consistent internally and with code. We shouldn't have silos of tests that use terms we used three years ago, and those we used a year ago, and so on. Going back and updating tests is difficult to sell to busy teams, but going back to update documentation after a big change is expected. So let's not talk about folders filled with hundreds of tests, let's talk about *evolving a living documentation system*. That makes it much easier to explain why things should be self-explanatory, why business users need access to this as well, and why it has to be nicely organized so that things are easy to find.

So there it is: I chose the names not because of previous popularity but because they make sense. The names for these process patterns should create a mental model that actually points out the important things and reduces the confusion. I hope that you'll see this and adopt this new terminology as well.