Gojko Adzic

Specification by Example

How successful teams deliver the *right* software

MANNING

Table of Contents

2

Key process patterns

Specification by Example is a set of process patterns that facilitate change in software products to ensure the right product is delivered effectively. The key patterns that are commonly shared by the most successful teams that I interviewed in researching this book, along with their relationships, are shown in figure 2.1. Most of the teams implemented new process ideas by trial and error during their search for ways to build and maintain software more efficiently. By revealing the patterns in their processes, I hope to help others implement these ideas deliberately.

Why patterns?

The process ideas presented in this book make up patterns in the sense that they are recurring elements used by different teams; I am not referring to Christopher Alexander's pattern definitions. Process ideas that I have cited occur in several different contexts and produce similar results. I haven't documented the forces and changes expected in more traditional pattern books. Due in part to the case studies in this book, the Agile Alliance Functional Testing Tools group organized several pattern-writing workshops to document and build a catalog of patterns in a more traditional sense, but this work will take some time to complete. I've decided to leave expanding the patterns into a more traditional format for future editions of this book.

In *Bridging the Communication Gap*, I focused mostly on the tangible outputs of Specification by Example, such as specifications and acceptance tests. I neglected to consider that teams in various contexts might need radically different approaches to produce the same artifacts. In this book, I focus on process patterns, how artifacts are created, and how they contribute to later artifacts in the flow.

Just-in-time

Successful teams don't implement the entire sequence at one time or for all the specifications, as shown in figure 2.1—especially not before development starts. Instead, teams derive the scope from goals once a team is ready for more work, for example, at the beginning of a project phase or a milestone. They proceed with specifications only when the team is ready to start implementing an item, such as at the start of the relevant iteration. Don't mistake the sequence in figure 2.1 for big Waterfall specifications.

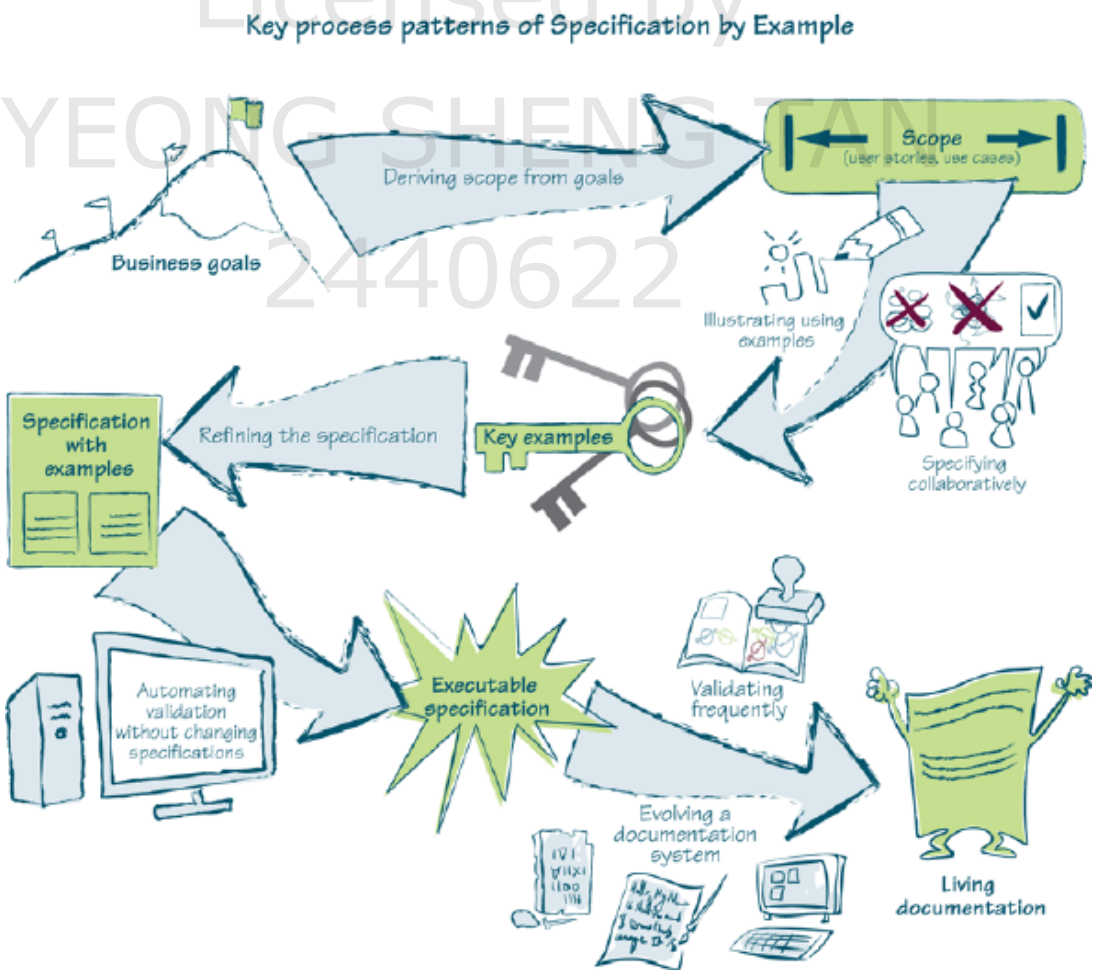

Figure 2.1 The key process patterns of Specification by Example

In this chapter, I present a brief overview of the key process patterns. Then we'll go over the key challenges and ideas for implementing each of these patterns in different contexts in part 2.

Deriving scope from goals

Implementation scope offers a solution to a business problem or a means to reach a business goal. Many teams expect a customer, a product owner, or a business user to decide on the scope of work before the implementation starts (everything that occurs before the implementation is often ignored by the software development team). After business users specify exactly what they want, software delivery teams implement it. This is supposedly what will make the customers happy. In fact, this is when issues with building the right product begin.

By relying on customers to give them a list of user stories, use cases, or other relevant information, software delivery teams are asking their customers to design a solution. But business users aren't software designers. If the customers define the scope, the project doesn't benefit from the knowledge of the people in the delivery team. This results in software that does what the customer asked but not what they really wanted.

Instead of blindly accepting software requirements as a solution to an unknown problem, successful teams *derive scope from goals*. They begin with a customer's business goal. Then they collaborate to define the scope that will achieve that goal. The team works with the business users to determine the solution. The business users focus on communicating the intent of the desired feature and the value they expect to get out of it. This helps everyone understand what's needed. The team can then suggest a solution that's cheaper, faster, and easier to deliver or maintain than what the business users would come up with on their own.[1]

Specifying collaboratively

If developers and testers aren't engaged in designing specifications, those specifications have to be separately communicated to the team. In practice, this leaves many opportunities for misunderstanding; details can get lost in translation. As a consequence, business users have to validate the software after delivery, and teams have to go back and make changes if it fails validation. This is all unnecessary rework.

Instead of relying on a single person to get the specifications right in isolation, successful delivery teams collaborate with the business users to specify the solution. People coming from different backgrounds have different ideas and use their own experienced-based techniques to solve problems. Technical experts know how to make better use

[1] For some good examples, see http://gojko.net/2009/12/10/challenging-requirements

of the underlying infrastructure or how emerging technologies can be applied. Testers know where to look for potential issues, and the team should work to prevent those issues. All this information needs to be captured when designing specifications.

Specifying collaboratively enables us to harness the knowledge and experience of the whole team. It also creates a collective ownership of specifications, making everyone more engaged in the delivery process.

Illustrating using examples

Natural language is ambiguous and context dependent. Requirements written in such language alone can't provide a full and unambiguous context for development or testing. Developers and testers have to interpret requirements to produce software and test scripts, and different people might interpret tricky concepts differently.

This is especially problematic when something that seems obvious actually requires domain expertise or knowledge of jargon to be fully understood. Small differences in understanding have a cumulative effect, often leading to problems that require rework after delivery. This causes unnecessary delays.

Instead of waiting for specifications to be expressed precisely for the first time in a programming language during implementation, successful teams *illustrate specifications using examples.* The team works with the business users to identify *key examples* that describe the expected functionality. During this process, developers and testers often suggest additional examples that illustrate edge cases or address areas of the system that are particularly problematic. This flushes out functional gaps and inconsistencies and ensures that everyone involved has a *shared understanding* of what needs to be delivered, avoiding rework that results from misinterpretation and translation.

If the system works correctly for all the key examples, then it meets the specification that everyone agreed on. Key examples effectively define what the software needs to do. They're both the target for development and an objective evaluation criterion to check to see whether the development is done.

If the key examples are easy to understand and communicate, they can be effectively used as unambiguous and detailed requirements.

Refining the specification

An open discussion during collaboration builds a shared understanding of the domain, but resulting examples often feature more detail than is necessary. For example, business users think about the user-interface perspective, so they offer examples of how something should work when clicking links and filling in fields. Such verbose descriptions constrain the system; detailing how something should be done rather than what is required is wasteful. Surplus details make the examples harder to communicate and understand.

Key examples must be concise to be useful. By *refining the specification,* successful teams remove extraneous information and create a concrete and precise context for development and testing. They define the target with the right amount of detail to implement and verify it. They identify what the software is supposed to do, not how it does it.

Refined examples can be used as acceptance criteria for delivery; development isn't done until the system works correctly for all examples. After providing additional information to make key examples easier to understand, teams create specifications with examples, which is a specification of work, an acceptance test, and a future functional regression test.

Automating validation without changing specifications

Once a team agrees on specifications with examples and refines them, the team can use them as a target for implementation and a means to validate the product. The system will be validated many times with these tests during development to ensure that it meets the target. Running these checks manually would introduce unnecessary delays, and the feedback would be slow.

Quick feedback is an essential part of developing software in short iterations or in flow mode, so we need to make the process of validating the system cheap and efficient. An obvious solution is automation. But this automation is conceptually different from the usual developer or tester automation.

If we automate the validation of the key examples using traditional programming (unit) automation tools or traditional functional-test automation tools, we risk introducing problems if details get lost between the business specification and technical automation. Technically automated specifications will become inaccessible to business users. When the requirements change (and that's *when*, not *if*) or when developers or testers need further clarification, we won't be able to use the specification we previously automated. We could keep the key examples both as tests and in a more readable form, such as Word documents or web pages, but as soon as there's more than one version of the truth, we'll have synchronization issues. That's why paper documentation is never ideal.

To get the most out of key examples, successful teams automate validation without changing the information. They literally keep everything in the specification the same during automation—there's no risk of mistranslation. As they *automate validation without changing specifications*, the key examples look nearly the same as they did on a whiteboard: comprehensible and accessible to all team members.

An automated Specification with Examples that is comprehensible and accessible to all team members becomes an *executable specification.* We can use it as a target for development and easily check if the system does what was agreed on, and we can use that same document to get clarification from business users. If we need to change the specification, we have to do so in only one place.

If you've never seen a tool for automating executable specifications, this might seem unbelievable, but look at figure 2.2 and figure 2.3. They show executable specifications fully automated with two popular tools, Concordion and FitNesse.

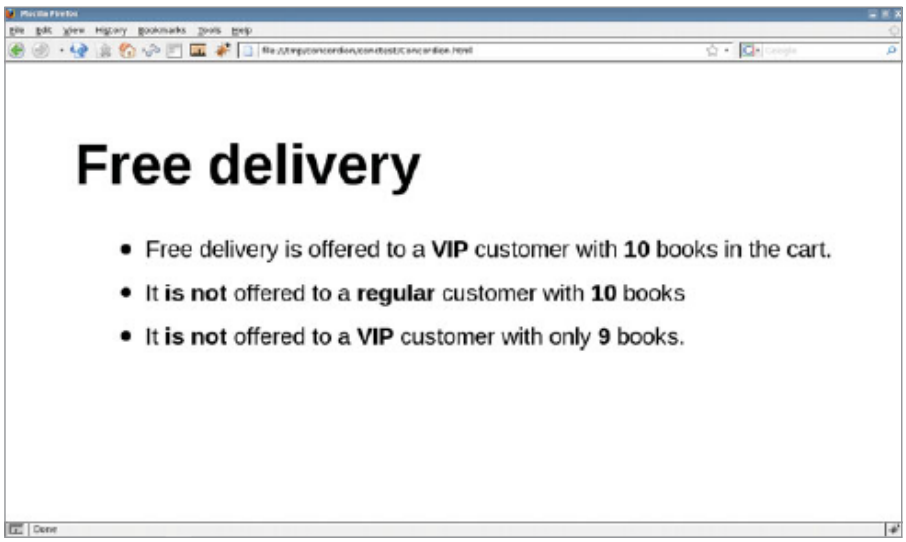

Figure 2.2 An executable specification automated with Concordion

The prize pool is divided among the winners using the following distribution for winning combinations (number of correct hits out of six chosen numbers). Example below is for $2M payout pool.

Prize Distribution for Payout Pool	2,000,000	
Winning Combination	Pool Percentage?	Prize Pool?
6	68	1,360,000
5	10	200,000
4	10	200,000
3	12	240,000

Figure 2.3 An executable specification automated with FitNesse

Many other automation frameworks don't require any translation of key examples. This book focuses on the practices used by successful teams to implement Specification by Example rather than tools. To learn more about the tools, visit http://specificationbyexample.com, where you will be able to download articles explaining the most popular tools. See also the "Tools" section in the appendix for a list of suggested resources.

Tests are specifications; specifications are tests

When a specification is described with very concrete examples, it can be used to test the system. After such specification is automated, it becomes an executable acceptance test. Because I write about only these types of specifications and tests in this book, I'll use the words *specifications* and *tests* interchangeably; for the purposes of this book, there's no difference between them.

This doesn't mean there aren't other kinds of tests—for example, exploratory tests or usability tests are not specifications. The context-driven testing community tries to distinguish between these classes of tests by using the name checks for deterministic validations that can be automated and tests for non-deterministic validations that require human opinion and expert insight.[†] In the context-driven language, this book addresses only designing and automating checks. With Specification by Example, testers use expert opinion and insight to design good checks in collaboration with other members of the team. Testers don't execute these checks manually, which means they have more time for other kinds of tests.

[†] See www.developsense.com/blog/2009/08/testing-vs-checking

Validating frequently

In order to efficiently support a software system, we have to know what it does and why. In many cases, the only way to do this is to drill down into programming code or find someone who can do that for us. Code is often the only thing we can really trust; most written documentation is outdated before the project is delivered. Programmers are oracles of knowledge and bottlenecks of information.

Executable specifications can easily be validated against the system. If this validation is frequent, then we can have as much confidence in the executable specifications as we have in the code.

By checking all executable specifications frequently, teams I talked to quickly discover any differences between the system and the specifications. Because executable specifications are easy to understand, the teams can discuss the changes with their business users and decide how to address the problems. They constantly synchronize their systems and executable specifications.

Evolving a documentation system

The most successful teams aren't satisfied with a set of frequently validated executable specifications. They ensure that specifications are well organized, easy to find and access, and consistent. As their projects evolve, the teams' understanding of the domain changes. Market opportunities also cause changes to the business domain models. The teams that get the most out of Specification by Example update their specifications to reflect those changes, evolving a living documentation system.

Living documentation is a reliable and authoritative source of information on system functionality that anyone can access. It's as reliable as the code but much easier to read and understand. Support staff can use it to find out what the system does and why. Developers can use it as a target for development. Testers can use it for testing. Business analysts can use it as a starting point when analyzing the impact of a requested change of functionality. It also provides free regression testing.

A practical example

In the rest of this book, I'll focus on process patterns rather than artifacts of the process. To put things into perspective and ensure that you understand these terms, I've included an example of artifacts produced during the entire process, from business goals to the living documentation system. The discussion of the example indicates the chapters in which I'll discuss each part of the process.

Business goal

A business goal is the underlying reason for a project or project milestone. It's the guiding vision that got the business stakeholders, internal or external, to decide to invest money in software development. Commercial organizations should be able to clearly see how such goals can either earn, save, or protect money. A good start for a business goal could be "Increase repeat sales to existing customers." Ideally, a goal should be measurable, so it can guide the implementation. The right software scopes for "Increase repeat sales to existing customers by 10% over the next 12 months" and "Increase repeat sales to existing customers by 500% over the next 3 months" are most likely very different. A measurable goal makes it possible to ascertain whether the project succeeded, to track progress, and to prioritize better.

An example of a good business goal

Increase repeat sales to existing customers by 50% over the next 12 months.

Scope

By applying the practices that I'll describe in chapter 5, we derive the implementation scope from the business goals. The implementation team and the business sponsors come up with ideas that can be broken down into deliverable software chunks.

Let's say we identify a theme for a customer loyalty program that can be broken down into basic loyalty system features and more advanced bonus schemes. We decide to focus on building a basic loyalty system first: customers will register for a VIP program, and VIP customers will be eligible for free delivery on certain items. We'll postpone any discussion on advanced bonus schemes for later. Here's the scope for this example:

User stories for a basic loyalty system

- In order to be able to do direct marketing of products to existing customers, as a marketing manager I want customers to register personal details by joining a VIP program.
- In order to entice existing customers to register for the VIP program, as a marketing manager I want the system to offer free delivery on certain items to VIP customers.
- In order to save money, as an existing customer I want to receive information on available special offers.

Key examples

By applying the practices described in chapters 6 and 7, we produce detailed specifications for the appropriate scope once our team starts implementing a particular function. For example, when we start working on the second item of the scope—free delivery—free delivery must be defined. During collaboration, we decide that the system will offer free delivery on books only, to avoid logistical problems related to shipping electronics and large items. Because the business goal is to promote repeat sales, we try to get customers to buy several items; "free delivery" becomes "free delivery for five or more books." We identify key examples, such as a VIP customer buying five books, a VIP customer buying fewer than five books, or a non-VIP customer buying books.

This leads to a discussion about what to do with customers who purchase both books and electronics. Some suggestions relate to expanding the scope: for example, splitting the order in two and offering free delivery for the books only. We decide to postpone this option and implement the simplest thing first. We won't offer free delivery if there is anything other than books in the order. We add another key example to the current set, to be revisited later:

Key Examples: Free delivery

- VIP customer with five books in the cart gets free delivery.
- VIP customer with four books in the cart doesn't get free delivery.
- Regular customer with five books in the cart doesn't get free delivery.
- VIP customer with a five washing machines in the cart doesn't get free delivery.
- VIP customer with five books and a washing machine in the cart doesn't get free delivery.

Specification with examples

By applying the practices from chapter 8, we refine the specification from the key examples and create a document that's self-explanatory and formatted in a way that will make it easy to automate the validation later (as shown below):

Free delivery

- Free delivery is offered to VIP customers once they purchase a certain number of books. Free delivery is not offered to regular customers or VIP customers buying anything other than books.
- Given that the minimum number of books to get free delivery is five, then we expect the following:

Examples

Customer type	Cart contents	Delivery
VIP	5 books	Free, Standard
VIP	4 books	Standard
Regular	10 books	Standard
VIP	5 washing machines	Standard
VIP	5 books, 1 washing machine	Standard

This specification—a self-explanatory document—can be used as a target for implementation and as a driver for an automated test so we can objectively measure when the implementation is done. It's stored in a repository of specifications, to become part of the living documentation. An example would be a FitNesse wiki system or a directory structure of Cucumber feature files.

Executable specification

When our developers start working on the feature described in the specification, the test based on this specification will initially fail because it's not yet automated and the feature isn't yet implemented.

The developers will implement the relevant feature and connect it to the automation framework. They'll use an automation framework which pulls the inputs from the specification and validates the expected outputs without requiring them to actually change the specification document. The ideas and practices in chapter 9 will help automate the specification efficiently. Once the validation is automated, the specification becomes executable.

Living documentation

All the specifications for all implemented features will be validated frequently, most likely by an automated build process. This helps to prevent functional regression issues while ensuring that specifications stay current. The team will use the practices from chapter 10 so that frequent validation goes smoothly.

When the entire user story is implemented, someone will first validate that it's done and then restructure the specifications so that they fit in with the specifications for features that were already implemented. They'll use the practices from chapter 11 to evolve a documentation system from the specifications in increments. For example, they might move the specification for free delivery into the hierarchy of features related to delivery, potentially merging them with other free-delivery examples triggered by different factors. In order to make the documentation easier to access, they might set up links between the specification for free delivery and the specifications for other delivery types.

Then the cycle starts again. Once we need to revisit the rules for free delivery—for example, when working on the advanced bonus schemes or in order to extend the functionality to split orders with books from orders with other items—we'll be able to use the living documentation to understand the existing functionality and specify changes. We can use the existing examples to make specifying collaboratively and illustrating using examples more effective. We'll then produce another set of key examples, which will lead to an increment of the specification for free delivery that will ultimately be merged with the rest of the specifications. And the cycle will repeat.

Now that we have had a quick overview of key process patterns, we'll take a closer look at living documentation in chapter 3. In chapter 4, I present ideas on how to start adopting Specification by Example, followed by ideas on implementing individual process patterns in part 2.

Remember

- The key process patterns of Specification by Example are deriving scope from goals, specifying collaboratively, illustrating specifications using examples, refining the specifications, automating validation without changing the specifications, validating the system frequently, and evolving living documentation.
- With Specification by Example, functional requirements, specifications, and acceptance tests are the same thing.
- The result is a living documentation system that explains what the system does and that is as relevant and reliable as the programming language code but much easier to understand.
- Teams in different contexts use different practices to implement process patterns.