Kamil Nicieja
Foreword by Gojko Adžić

Writing Great Specifications

Using specification by example and Gherkin

Sample Chapter

MANNING

Writing Great Specifications

by Kamil Nicieja

Chapter 1

brief contents

1 Introduction to specification by example and Gherkin

This chapter covers

- Examining why teams need specifications
- Recognizing common specification pitfalls
- Understanding the basics of specification by example and Gherkin
- Solving common delivery problems with specification by example and Gherkin

How well we communicate is determined not by how well we say things, but how well we are understood.

—Andy Grove

The money is all on the right [side of the product life cycle], in the area of certainty [where the product is mature]. I work on the left, with uncertainty. I'll never be rich.

—Chris Matts

Humanizing technology is perhaps *the* greatest challenge of software engineering. The technology industry must strive to show tremendous empathy for other people's problems. We're making tools for everyone out there. In the messy world of organizational politics, broken workflows, human errors, and biases, technology experts must figure out how to successfully deliver great software. It's an important responsibility.

To do our job well, we have to

- Make sure we deliver *the right software*
- Deliver it *the right way*

Delivery teams are naturally competent in delivering software the right way. As an industry, we've developed tools, standards, and methodologies that make our designs beautiful and usable—and our code performant, secure, and easy to maintain. We keep getting better at refining and reinventing our best practices.

"The right software" part, though … what does that even mean? Every time I explain to someone what this book is about, I tell them that, as programmers, we're taught to *write code*—algorithms, design patterns, abstractions. Or, as designers, we're taught to *design*—organize information and create beautiful interfaces. But when we get our entry-level jobs, our employers expect us to "deliver value to our clients." And the client may be, for example, a *bank*. If I'd been working for a bank when I started my career, it would have quickly come up that I know next to nothing about banking—except how to efficiently decrease my account balance. So I would have had to somehow *translate* what was expected of me into code. I would have needed to build a bridge between banking and my technical expertise if I wanted to deliver any value. "This," I say, "is what the book is about: building bridges between technology and business." Over the course of multiple projects I've had the privilege to work on, I've come to believe that these bridges can only be built with empathy—understanding other people's problems—and inclusive communication.

Even though engineers should be good at building bridges, our industry seems to have a problem with delivering *the right software*. In practice, delivering the right software requires securing the right requirements. I'll talk more about requirements in a moment. For now, I'll say the following:

- A 1994 study showed that 31.1% of projects were canceled before they were completed, and 52.7% of projects cost 189% of their original estimates.[1]
- In larger companies, rare successful projects had only 42% of the originally proposed features.[2]
- In 2000, IBM and Bell Labs studies showed that 80% of all product defects are inserted at the requirements-definition stage.[3]

---

[1] The Standish Group, "The CHAOS Report" (1995), http://mng.bz/40M3.

[2] Ibid.

[3] Ivy Hooks and Kristin Farry, *Customer-Centered Products: Creating Successful Products Through Smart Requirements Management* (AMACOM/American Management Association, 2001).

- Requirements errors consume from 28% to more than 40% of a typical project's budget.[4]
- Requirements defects account for the vast majority of the total cost of all defects—often 70% or more.[5]
- In 2008, almost 70% of companies surveyed set themselves up for both failure and significantly higher costs by their use of poor requirements practices.[6]

What are the consequences? Commercial organizations across the European Union lost €142 billion on failed IT projects in 2004 alone, mostly because of poor alignment with business objectives or business strategies becoming obsolete during delivery.[7] So although we're pretty good at maintaining our technical standards of excellence, we apparently still have a lot to learn when it comes to understanding what businesses need from us.

In this chapter and throughout the book, I'll introduce you to a selection of bridge-building methods for translating business objectives into working software that, in my experience, results in great and meaningful products and services. This chapter will begin your in-depth journey of learning to write *executable specifications in Gherkin* according to the key practices of *specification by example*.

Specification by example (SBE) is a collaborative software development approach that facilitates collaboration by illustrating software requirements with concrete examples and automated acceptance tests. Because SBE is a process, you'll need some tools that will help you implement that process. This is why you're going to learn *Gherkin*. Gherkin is a business-readable, domain-specific language that's easy for nontechnical folks to understand. As such, it makes translating requirements into code easier.

In a way, the book is an advanced Gherkin tutorial with some product-design ambitions. I'll talk more about the reasons for choosing Gherkin later in the chapter. But when I was first learning SBE's key patterns, I found that, although locating material on automated acceptance tests and eliciting better requirements is easy, there aren't many resources available on writing great executable specifications. By *great*, I mean well-written and easy to read in terms of sentences and words, not code. That makes my ambition small, because I chose a specific topic for the book. I care about making sure that well-elicited requirements aren't misrepresented by poorly written specifications. At the same time, I realize that writing executable specifications is a cross-disciplinary matter. Whenever I can, I'll talk about making your requirements better and more specific with clever Gherkin techniques. Other times, I'll point you toward specific books that talk about requirements, product design, or marketing, in hopes they will answer your further questions.

---

[4] Ibid.

[5] Dean Leffingwell and Don Widrig, *Managing Software Requirements: A Use Case Approach*, 2nd ed. (Addison-Wesley Professional, 2003).

[6] IAG Consulting, "Business Analysis Benchmark" (2008).

[7] Gojko Adžić, *Impact Mapping* (Provoking Thoughts, 2012).

This chapter offers an overview of what a specification is and how SBE and Gherkin fit into the software development landscape. If you're a non-engineer, you'll learn how to make essential contributions to automated testing without having to learn to write testing code. (Don't worry about technical lingo. I use it rarely and explain it when I do.) Engineers and testers will find SBE and Gherkin helpful in striking a stronger chord with nontechnical audiences through automated specifications. You'll also begin to see SBE as a single process to guide product development through requirements analysis, design, development, testing, and so on.

1.1 What's a specification?

Imagine that you and the team you work with have been brought in to work on a new version of a management system for a local public transport company. To get on with work, you need a list of functionalities, user stories, blueprints, sketches—anything that will let you write some code or make a UI mock-up. You need a specification.

DEFINITION *Specification*—An analysis of a system and its design, made to plan and execute the implementation

The word *specification* can mean a written document or an act of specifying. You'll see that I switch freely between both meanings. Whenever it's important to make a clear-cut distinction, I'll use a term like *specification document* or *specification process*. But you can assume that most of the time, I have the broad meaning of the word in mind.

In the case of the example public transport company, to devise a specification, you have to agree on a list of requirements and functionalities the new release must satisfy.

DEFINITION *Requirement*—A capability or condition that must be met or possessed by a solution to satisfy market needs or a contract, a standard, a specification, or other formally imposed documents

For example, you and the business owners may agree that a good requirement would be to apply discounts when students or retirees buy tickets. Other examples could relate to handling season tickets, performing online payments, managing customers, and reports.

Delivery teams can write down their requirements in a functional requirements document, but they may also encapsulate requirements in use cases, which are shorter, or use user stories as tickets for a future in-depth conversation about the requirements. The final method depends on the software development process chosen by the team.

1.2 Why do teams need specifications?

Traditionally, specifications have had a bad reputation in the software development community. The reason is half psychological, half practical.

Psychologically, specifications seem to promise the same success as following a cooking recipe. They invite a "Follow the steps, and everything will be all right" mindset. The

promise is as reassuring as it is deceiving. In practice, creating a complete specification is extremely difficult, if not impossible.

No software development team functions without specifications, though. Whether you write an official document or have a casual conversation about the requirements during a workshop, you're still specifying.

The one and only reason teams need specifications is *information asymmetry*. Teams need to distribute information evenly among the stakeholders to create the best possible product. If they don't, they'll miss critical requirements and make an incomplete product—or even a broken one.

DEFINITION *Information asymmetry*—A situation in which one party has more or better information than another

To reduce information asymmetry, teams create specifications—recipes defining *what* needs to be done or *how* it needs to be done. Specifications can help fight information asymmetry in two ways:

- A specification can define acceptance criteria that help examine whether a team has delivered a complete system that works.
- A specification can provide a common language that allows technical and nontechnical stakeholders to understand each other when they talk about requirements.

We'll now go into more depth on both of these topics.

1.2.1 Defining acceptance criteria in specifications

Assume that you and the public transport company's management team have agreed that the system you're building should include two subsystems:

- An internal management application for updating bus schedules
- A mobile timetable application with journey-planning functionality

Sounds reasonable, doesn't it? The capabilities for both the employees of the company and its customers are clearly defined. But are they really?

Every time you analyze a requirement, you'll eventually stop talking about general capabilities of the system and start thinking in terms of concrete, discrete quality measures that the application must meet. When discussing our public transport company, I said that a good requirement would be to apply discounts when students or retirees buy tickets. But how can you determine whether that requirement is satisfied without going into more detail? For example, you'd need to declare that students can have a 30% discount and retirees can have a 95% discount. These two declarations would allow you to say that the requirement was in fact satisfied and implemented correctly. Such quality measures are called *acceptance criteria*.

DEFINITION *Acceptance criterion*—A condition or quality measure that a software product must meet to satisfy requirements

Acceptance criteria *illustrate* requirements. You should be able to use a criterion to evaluate the system and get an unambiguous confirmation that the system either passes or fails your test: for example, "A bus road should consist of at least two bus stops." Right, and that's how the system behaves. "Timetables for work weeks should be different than timetables for weekends." Oops, we forgot about that; let's go back to the drawing board. You should be able to get a binary response to every criterion—as in *yes* or *no* questions. Without that binary response, you can't say whether the system is complete and works as it should.

Raw requirements are often too difficult to comprehend without further analysis. Without clear acceptance criteria for each of the requirements, delivery teams can't plan any work ahead and deliver any value in a predictable way. When there's not a good specification, functionality usually suffers from rework or bugs that cause delays and cost a lot. Good acceptance criteria ensure that the implemented solution meets the demands of your stakeholders.

1.2.2 Building a ubiquitous language into specifications

Imagine for a moment that after you finish the beta version of the mobile journey planner, the customer support department receives a phone call from an angry customer:

> The customer begins, *"I downloaded the app to help me during my two-day stay in the city. But I can't get where I want!"*
>
> *"What street are you on? What's wrong?"*
>
> *"I've got a meeting in Edison. I used your app to get there, but I can't find the building I'm supposed to enter. It's all wrong!"*
>
> *"Wait—do you mean Edison Street or Edison Business Center? They're two different places."*

The customer wanted to plan the journey without knowing what street the destination building was on, but the application didn't support such a behavior. To add insult to injury, the mobile app chose Edison *Street*, located elsewhere in the city, as the final destination, because it couldn't find Edison *Business Center* in the database.

The result? The user and the application spoke two different languages, and the confused customer got lost. The dictionary of the developers who built the app was restricted to streets; after all, bus stops inherit their names from where they're located. That's how the system works, the team said. What they didn't know was that their customers don't think about the rules of a system—they only want to arrive on time.

To avoid similar mistakes, delivery teams should strive to grasp the language their users speak and align their language with this language. The result of this alignment is often called a *ubiquitous language*.

> **DEFINITION** *Ubiquitous language*—A common language between developers and domain experts

A ubiquitous language is "a language cultivated in the intersection of [technical and business] jargons."[8] The development of journey-planning software requires knowledge in two different domains: journey planning and software. Experts in both areas must communicate understandably.

DEFINITION *Domain*—What an organization does, and the world it does it in

The journey planners will use the jargon of their field and have limited understanding of the technical dictionary of software development. Developers, on the other hand, will understand and discuss the system in terms such as objects, methods, and design patterns. Having a single common language eliminates the cost of mental translation and reduces the number of misunderstandings—the ratio of noise in the signal—in discussions between technical and nontechnical stakeholders. Translation blunts communication and makes domain learning anemic.

The journey planners from the example can also be called *domain experts*. Domain experts help you create a ubiquitous language. When either the business side or the technical side discovers a misunderstanding, they can use the opportunity to improve their shared dictionary and avoid the same mistake the next time. This way, they build a shared *domain model*, which will improve in quality over time.

DEFINITION *Domain expert*—A person who is an authority in a particular area or topic. The term usually refers to a domain other than the software domain.

DEFINITION *Domain model*—A simplification of the real-world business domain. It's an interpretation of reality that abstracts the aspects relevant only to solving the problem at hand.

The ubiquitous language fuels the domain model. Having a shared dictionary of important business concepts creates a platform for discussing data, behaviors, and relationships within the model in a meaningful way, with a certainty that everybody is on the same page. In the journey-planning example, the team thought that a *destination* was the same as a *street*; but it turned out that users assumed there are other kinds of destinations, such as *buildings* and *points of interest*. Having established a baseline, the team can use the common language to establish clear relationships between the concepts of destinations, streets, buildings, and points of interests.

A specification can help develop the ubiquitous language. It's a container where all important domain concepts can be stored after they're encountered and analyzed by the team. When that happens, and the process is thorough and successful, the specification becomes a documentation of the domain, the knowledge base of the delivery team. When a specification fails to contribute to the ubiquitous language or doesn't create a truthful domain model, the team may misunderstand requirements, which often leads to expensive rework.

---

[8] Eric Evans, *Domain-Driven Design* (Addison-Wesley, 2003).

1.3 Common specification pitfalls

Much of software engineering is about building systems right, but specifications, requirements, and acceptance criteria are about building the right system. From time to time, every software engineer experiences a painful push-back caused by a sloppy analysis of the requirements. You, too, know what's at stake. This section should help you identify some pitfalls you yourself may have encountered.

I want to discuss these five anti-patterns:

- Over-specification
- Hand-offs
- Under-documentation
- Speci-fiction
- Test-inability

I named each anti-pattern in a distinctive way that will help you remember what it's about. Hopefully, as you go through the sections that follow, the names of the anti-patterns will become clearer to you, and I'll achieve my goal.

1.3.1 Over-specification

A popular first instinct meant to defend a project against ambiguity and insufficient planning is to try to design and plan as much as we can up front. I call that *over-specification*.

DEFINITION *Over-specification*—Doing too much specification up front

It's definitely easier to remove or change a requirement during an analysis phase; the more time we invest in implementing it, the more unmotivated we become when we have to kill it. The up-front approach aims to remove useless implementations, design flaws, and predictable errors as early as possible in exchange for a longer analysis phase. But software development teams must also understand that over-specification can lead to a state of *analysis paralysis*.

DEFINITION *Analysis paralysis*—A productivity block created in search of the perfect—unattainable—design

In extreme cases, bureaucratic or regulated environments may demand over-specification by requesting specification documents that can run into thousands of pages. (Bear in mind, though, that analysis paralysis isn't limited to written specifications.) But unless you're making software for surgeons, analyzing every single detail in advance often feels unnecessary—even harmful.

1.3.2 Hand-offs

Handing off requirements looks like a classic waterfall mistake—an artifact from the past—but I still see agile teams struggling with hand-offs, often due to their organization's internal politics. Any requirement can be handed off.

DEFINITION *Hand-off*—A situation in which somebody analyzes requirements without the input of the delivery team, signs off on the scope by writing down the analyzed requirements, and later hands off those requirements to the delivery team to complete

Hand-offs result in a fragmented communication flow between business and delivery. In my experience, people who hand off requirements are often business users, managers, analysts, product owners, or designers, depending on the chain of command in a given organization. In a management-oriented company, managers are more likely to create an environment where they can decide on the list of requirements and the scope, trying to maintain control over important decisions. I've seen the same thing happen with design teams in design-oriented organizations. And engineers, too, can hand off requirements if they're within their areas of expertise. (Think of technical, nonfunctional requirements such as performance, security, or low-level integrations.) Nobody's a saint.

Such organizations mistake the communication structure for the organizational structure. A company can be management-oriented, design-oriented, or engineering-oriented and still have a healthy, collaborative, and inclusive process.

Hand-offs cause various problems with delivery. A team that only receives a specification won't understand the context in which the requirements were collected. Their decision-making abilities will be impaired when it comes to split-second decisions. The team won't be able to make on-the-fly decisions because they won't know the thought process that led to making the requirements the way they are. They will only see the final result—the specification. They may also be too afraid to change anything. And in over-specified documents, contradictions and ambiguities can occur easily. When hand-offs like these happen, misunderstandings creep in and cause expensive rework to appear later in the process.

TIP Don't let documentation replace communication.

1.3.3 Under-documentation

Many delivery teams burnt by over-specification discard it in favor of an implementation-first approach, eradicating any up-front practices. An implementation-first approach optimizes for writing software without dealing with wasteful documentation and specifications. It rejects huge design commitments before customers prove they want the solution—and the only way to prove it is to hack some code together and release it as soon as possible, rejecting any process that doesn't help write production code. For example, Extreme Programming advocates use no extra design documents and let the code speak for itself. Running code doesn't lie, as a document might. The behavior of running code is unambiguous.

Initially, the implementation-first approach feels efficient, especially in young companies—but as the organization grows and the product matures, diseconomies of scale kick in. Not everyone is a coder. Communication and decision-making start causing

trouble, and adding new people to the team slows work instead of making it faster. I call such a specification anti-pattern *under-documentation*.

> **DEFINITION** *Under-documentation*—Discarding documentation and letting code speak for itself in order to speed up development

Underdocumented teams are left with no clear path to track decisions made in the past. Institutional memory suffers; when people who worked on implementation become unavailable, temporarily or permanently, they take their knowledge with them. Building long-term understanding within the company often requires additional facilitation. Many teams hurting from under-documentation realize its downsides too late when fixing the problem gets painful.

> **TIP** Don't let *agile* be an excuse to ignore documentation.

1.3.4 Speci-fiction

Documentation and specification artifacts grow obsolete easily. As your product evolves over time, requirements often evolve, flat-out change, or turn out to be poorly defined and have to be refined. Documentation and specifications, like all internally complex documents, are often too difficult to update on a regular basis without introducing some inconsistencies. Outdated and unwanted, they become *speci-fiction*. (Yes, I invented the word. No, I'm not a poet.)

> **DEFINITION** *Speci-fiction*—A specification that poses as a single source of truth but that can't be one because nobody cares to update it

If you've ever struggled with outdated documentation, you're already familiar with the phenomenon of speci-fiction. Sometimes documents are left outdated because of multiple last-minute changes. In this case, the *fiction* in speci-fiction is that a new reader would be led to falsely believe that the specification or documentation describes the entire system *as it is*, when the working system is, in fact, different, because the requirements were changed during the release frenzy. Speci-fiction is only an illusion of correctness—an illusion that occurs when no single, reliable source of truth exists.

1.3.5 Test-inability

The INVEST mnemonic for agile software projects is a common reminder of the characteristics of a good-quality product backlog item such as a user story (see table 1.1). Much of INVEST is beyond the scope of this discussion; I won't expand on the topic directly, but I already talked about such characteristics as *valuable* and *small* when I discussed the difference between the right delivery and the right software at the beginning of this chapter and when I talked about over-specification and long specification documents.

Table 1.1 The INVEST mnemonic

Letter	Meaning	Description
I	Independent	The story should be self-contained.
N	Negotiable	The story should leave space for discussion about its scope.
V	Valuable	The story must deliver value to the stakeholders.
E	Estimable	The delivery team should always be able to estimate the size of the story.
S	Small	The smaller the story, the easier it is to analyze and estimate correctly.
T	Testable	The story should support test development.

I'd like to focus on the testability part, which many teams overlook. I've met many programmers and testers who, when working on a user story, weren't sure where to start, what to test and what not to test, how much to test in one go, what to call their tests, and how to understand why a test fails.

According to INVEST, testability should be baked into a good user story, because testability lays the foundation for quality. How can you be sure that you delivered any business value if you don't know how to test its implementation? Or how can you know that you'll continue to deliver value in the future, regardless of any system changes or errors? What I call *test-inability* is a team's failure to answer questions like these—a failure that originates in a bad specification process.

DEFINITION *Test-inability*—Lacking clear measures of value that can support development

1.4 Meet specification by example and Gherkin

Delivery teams choose the implementation-first approach despite its shortcomings because it gives them the freedom, agility, and productivity they love. On the other hand, the up-front approach has the upper hand in consistently producing somewhat reliable documentation. Is there any method that combines the best of both worlds? Fortunately, yes. Of the many tools and methodologies introduced by the community to reshape traditional specification methods, I find two particularly interesting and explore them in the book: SBE and Gherkin.

Specification by example, a set of practices that sprang from the agile acceptance-testing tree, is a collaborative approach to defining software requirements based on illustrating executable specifications with concrete examples. It aims to reduce the level of abstraction as early in the process as possible, getting everyone on the same page and reducing future rework.

Gherkin, a business-readable domain-specific language, provides a framework for business analysis and acceptance testing. Gherkin helps you understand requirements from the perspective of your customers. By forcing you to think about what a user's workflow will look like, Gherkin facilitates creating precise acceptance criteria. The

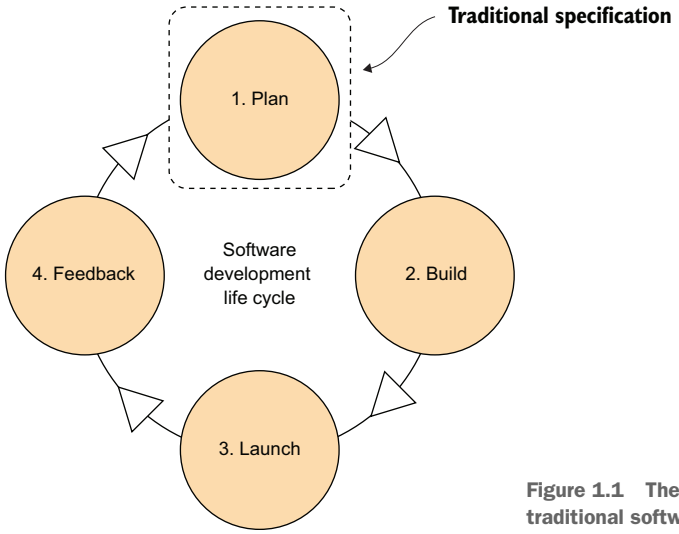

Figure 1.1 The place of specifications in the traditional software development process

book uses a Cucumber version of Gherkin's syntax. If you don't yet know what that means, don't worry—I'll explain everything in chapter 2.

SBE and Gherkin reimagine the traditional software development process. Every software development process follows similar phases as functionality progresses from conception to release (see figure 1.1). In most agile software development methodologies, the phases are as follows:

- Planning implementation
- Building the product
- Launching the product
- Getting feedback

Many teams also fall into a trap of treating specifying as a one-time *activity* that occurs during the planning phase, instead of as a *process* that keeps occurring as requirements evolve and change, which they often do throughout development. Teams that don't treat specification as a long-term process often behave like *automata*—machines designed to automatically follow a predetermined sequence of operations. In such a case, the sequence is defined during the planning phase and must be followed as long as no problems occur. But when a problem *does* occur, it's often already too late.

With SBE and Gherkin, as shown in figure 1.2, we follow a different paradigm. This paradigm requires us to use practices that must be performed throughout the entirety of a project—from analysis to maintenance. You'll see why when I talk more about *designing acceptance tests* (a testing activity) and *building living documentation* (a maintenance activity). Instead of creating a static document with requirements, I'll talk about a system of dynamic specification documents that *constantly evolves* along with the product.

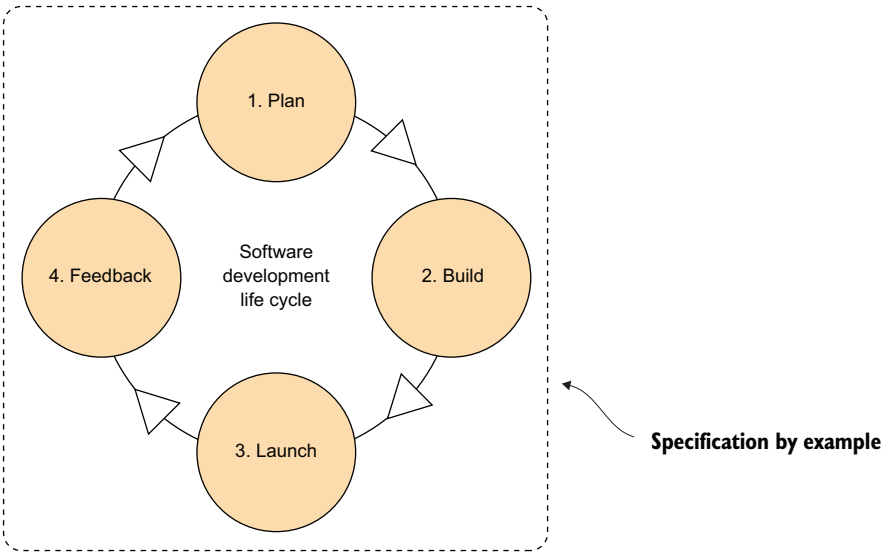

Figure 1.2 SBE reimagines the software development process by prolonging the specification process so that it takes place throughout the entire project.

If you're curious about what a specification written in Gherkin looks like, look at this example:

```
Feature: Setting starting points and destinations

  Scenario: Starting point should be set to current location

    Given a commuter that enabled location tracking
     When the commuter wants to plan a journey
     Then the starting point should be set to current location

  Scenario: Commuters should be able to choose bus stops and locations

     Given a bus stop at Edison Street
       And a Edison Business Center building at Main Street
      When the commuter chooses a destination
      Then the commuter should be able to choose Edison Street
       But the commuter should be also able to choose Edison Business Center
```

In order to help you write specifications like this, the upcoming chapters will apply SBE's key process patterns to Gherkin. You'll be able to offer programmers, designers, and managers an inclusive environment for clear communication, discovering requirements, and building a documentation system.

1.4.1 Key process patterns

Teams that apply SBE successfully introduce seven process patterns into their workflow.[9] In an SBE process that uses Gherkin—which, as you'll see later, is only one of

[9] Gojko Adžić, *Specification by Example* (Manning, 2011).

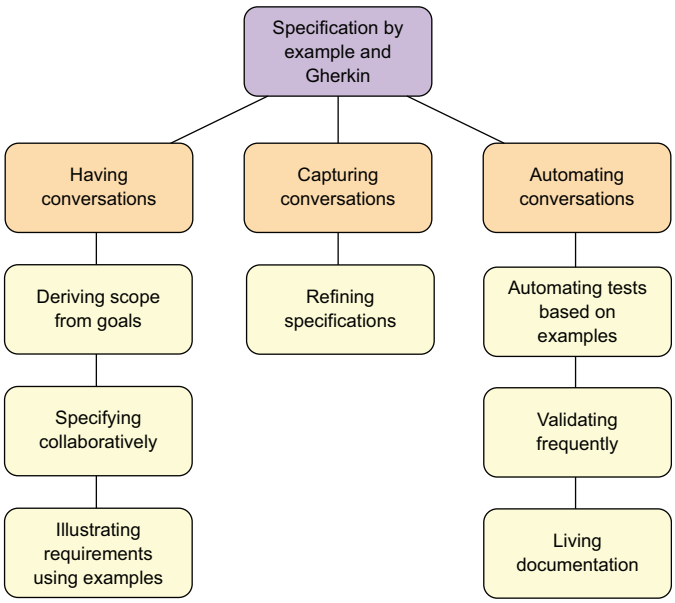

Figure 1.3 A high-level look at SBE's process patterns

several ways of applying SBE—these seven patterns can be split into three distinct groups revolving about the central concept of *conversations* (see figure 1.3).

Patterns focused on *having conversations* aim to increase the knowledge flow between the delivery team and the business as well as within the delivery team, without sacrificing agility. Patterns that deal with *automating conversations* ensure that the specifications stay up to date throughout the project's life cycle, allowing nontechnical stakeholders to check whether the use cases they care about work well within the system.

Capturing conversations links analysis and automation. Having conversations can't be a separate development activity, just as you can't write automated tests for the sake of writing tests. That's where the real magic begins, and where you'll meet Gherkin—it will let you write down your conversations in a form that's easy to automate.

1.5 Having conversations that identify business needs

The main premise of SBE and Gherkin is that frequent conversations between domain experts and the delivery team lay a foundation for the entire development process (see figure 1.4). Here are some examples of conversations:

- The public transport company's management wants to build new modules into their timetables system, and you discuss their business needs together.
- An angry customer explains that your mobile app shouldn't interpret Edison Business Center as Edison Street because they're not the same thing.

- Two engineers discuss whether the system should treat a bus route as a collection of 2D points on a map or a straight line between the start point and the destination point.
- A commuter files a bug report about the bus-scheduling functionality.
- You read customer feedback on social media and discover what new functionalities users want.

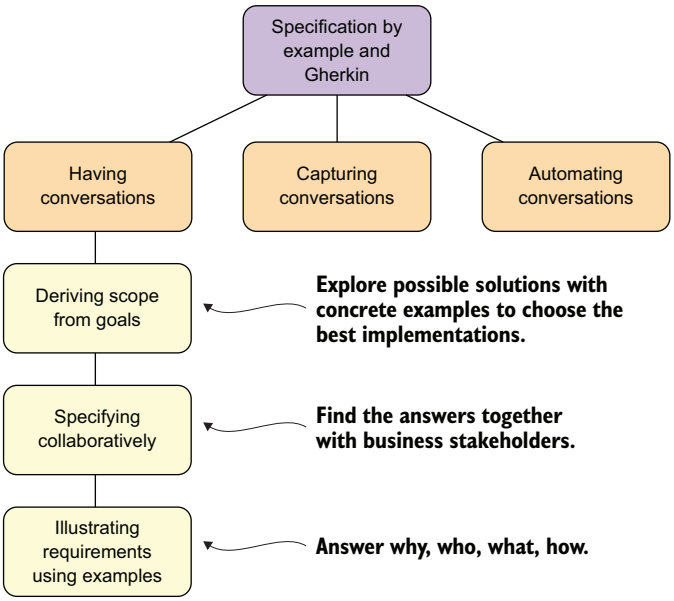

Figure 1.4 Having conversations should provide delivery teams with all the answers necessary to understand a project's goals, and who customers are and what solutions they need.

From these examples, we can reason that a *conversation* means a discussion between the business and the technology. Business domains and technology domains interact because they have to—if you want to create *any* software, let alone working software or, sometimes, even successful software, the team must understand the business context and have required technical excellence. The sections that follow analyze the topics that such interactions can follow.

1.5.1 Deriving scope from goals

Conversations typically revolve around four questions:

1. Why are we building this?
2. Who are we building this for?
3. What exactly are we going to build?
4. How will we build it?

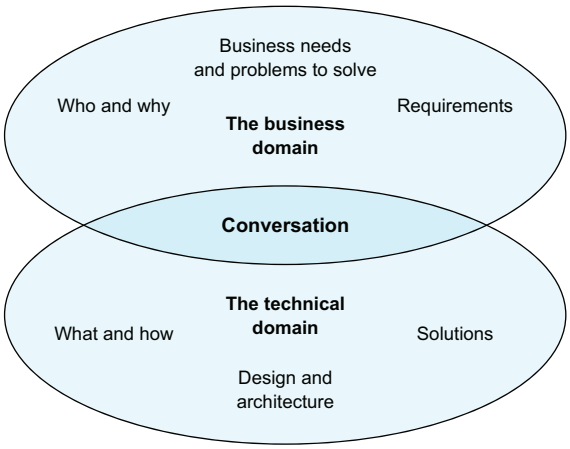

Figure 1.5 The business and technology domains must meet over the course of a conversation if you want to understand the business goals and set the optimal project scope.

Some answers come from the business domain and others from the technology domain (see figure 1.5). Usually, the business domain provides the *who* and the *why*, and the technology domain provides good *what* and *how* answers.

In general, answering questions at the top of the list will give you enough input to ask and answer the questions at the bottom. Such a practice—getting from business objectives to programmable solutions—is what SBE's practitioners call *deriving scope from goals*. Over the last five years, deriving scope from goals emerged as probably *the* most important practice in the modern landscape of software development.

Every major conference now features someone talking about the value of delivery people understanding business goals and designing software according to their company's objectives. Techniques such as impact mapping, feature injection, and user-story mapping have spread widely, changing the business analysis landscape. I, too, will talk about these techniques throughout the book.

The questions I listed help delivery teams understand why a solution is needed and who needs it. Answering them means discussing the company's goals and establishing success metrics. The goals and metrics, in turn, allow you to determine the scope of future work the team must deliver and build a framework that will let the team say whether they're making progress in terms of reaching their goals.

1.5.2 *Illustrating requirements with examples*

SBE and Gherkin require delivery teams to support their conversations with practical examples. *Illustrating requirements with examples* helps reduce the level of abstraction and leads to clearer acceptance criteria—especially if the examples are concrete instead of vague.

Humans prefer stories illustrated with examples. Say you were a lawyer who wanted to explain to your friend how splitting royalties works. If you said, "The writers should split the salary based on their contribution," your friend might not have a good idea

of what you meant. Each of you might understand the concept of "contribution" differently. But let's change that to "Here's an example: John, Gilly, and Robbie wrote a 250-page book together. John and Gilly wrote 100 pages each, so they should get 40% of the salary, because they each wrote 40% of the book—and Robbie, who wrote only 50 pages, should get 20% of the salary, just as 50 is 20% of 250." This time, your friend would probably grasp the full idea in a split second.

Clear storytelling invites good examples, because examples help us build better mental models of the new concepts we encounter. They're anchors. Links. Cognitive shortcuts. Most important, they reduce the likelihood of misunderstanding the purpose of a story. Requirements illustrated with good examples inherit all these benefits. They're simpler to digest and easier to keep in your head.

Let's look at a conversation without any concrete examples and a conversation full of examples to see if that's true. Here's the first conversation:

"Okay, so how should the application work?"

"I suppose that when commuters download our mobile app, they should be able to provide a starting point and a destination point, and see a timetable with all the bus lines and departure times they might find helpful in getting to the destination. It's very simple, really."

"Seems that way."

Such a conversation raises more questions than it answers. What are the starting points and destination points? Are they streets? Bus stops? Buildings and other places? And what exactly may a commuter "find helpful in getting to the destination?" There's no way we can know for sure.

What would happen, though, if we asked for concrete examples during the discussion?

"OK, so how should the application work?"

"Let's not jump to conclusions. Imagine for a moment that you're going to the city, say, on a business trip. How and when do you get there?"

"Well, I guess I might arrive a day earlier to be sure nothing goes wrong."

"So we're going to need a functionality to filter the timetables by date."

"Yes, we are. But let's consider what happens if the you're a bit more happy-go-lucky and arrive in the city an hour before the meeting. You don't have enough time to check where you are. Or maybe you don't know the exact street you must arrive at."

"Wow, we might need to implement a GPS geolocation functionality so we could help users know their current location."

"Yeah, and there should be an option to search for locations such as parks, buildings, and restaurants instead of only bus stop names."

"Seems that way."

Conversations with examples look similar to short stories about a system's behaviors. Good stories are vivid and build a platform for fertile discussion between the people

who read them. Bad stories confuse readers and leave people clueless. The same is true for good and bad specifications.

1.5.3 Specifying collaboratively

As you'll see in the sections to come, SBE and Gherkin redefine the distinction between analysis, design, and implementation by building a bridge between requirements and code. The practitioners should see the act of specifying as a process of continuous discovery through reducing their uncertainty about the requirements. Specifying is not a single activity or a phase to go through. In an agile process, requirements evolve as a project progresses because rarely does the knowledge exist up front to specify an application adequately.

Every time you have a conversation about your product, every time you ask a question about a requirement, every time you encounter a bug, every time you hear customer feedback—you're discovering whether your assumptions about the product are true or false. You're learning.

Sometimes, though, organized effort may be required to produce a reliable, repeatable specification process in a complex environment with multiple stakeholders. In such cases, SBE encourages specifying collaboratively by inviting the stakeholders to specification workshops or holding smaller, more regular meetings within the delivery team.

The participants should use the specification workshops to capture and refine good, concrete examples that emerged when the delivery team tried to derive scope from the business goals. They should then match the examples with requirements and acceptance criteria, letting the examples guide their analysis efforts.

Depending on the size of the team and the complexity of the product, specification workshops can range from multiday sessions featuring every important stakeholder to short, regular meetings between product owners, senior engineers, and designers. These workshops put a strong emphasis on knowledge sharing. Including diverse participants guarantees exploring multiple perspectives and covering different angles. Knowledge should flow freely within the team. Analysts, designers, developers, and testers should strive to understand what they're about to build, asking as many questions as they deem relevant. To achieve a common perspective on how customers will use the software, participants should learn the ubiquitous language of the business owners and the customers. Long story short, they should build a short-term understanding of the requirements that will guide their efforts in planning and during implementation.

> **WARNING** The topic of organizing and facilitating specification workshops, although important, is beyond the scope of the book, which focuses on writing skills. I only talk about workshops briefly in section 7.4. Chapter 7 is also where I mention a few resources and techniques for organizing workshops. For now, I advise you to read Gojko Adžić's original *Specification by Example*; chapter 6 talks about collaborative specification.

1.6 Long-term benefits of automating conversations

After the delivery team collects examples, team members create specifications out of conversations recorded in Gherkin. They automate the conversations and examples with software tests, validating the tests frequently to make sure the specifications stay up to date (see figure 1.6).

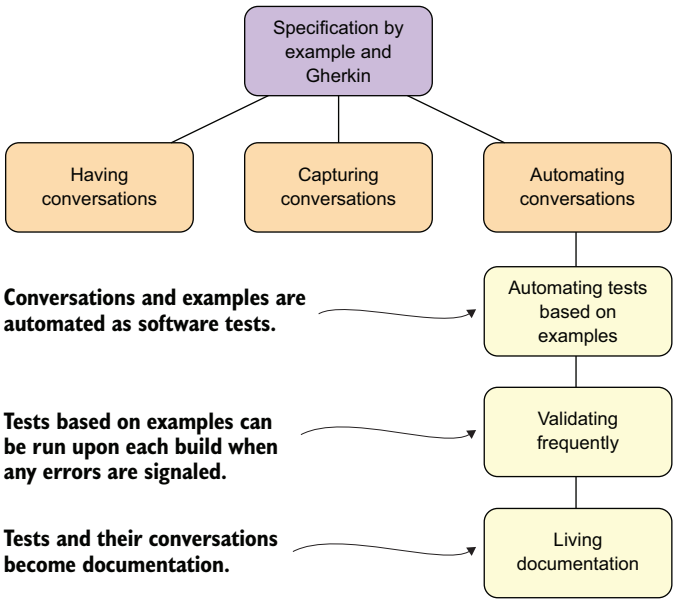

Figure 1.6 Automation turns conversations into executable test cases that, if validated frequently, become long-term system documentation.

I'll now talk about the elements of the automation process and why automating specifications gives delivery teams an enormous advantage. Don't be surprised that I haven't yet discussed recording conversations in Gherkin, even though the translation process is a prerequisite for automation. I want you to understand the benefits and challenges of team specification and automation first, so that you'll be free to draw your own conclusions when we explore Gherkin.

1.6.1 Automating tests based on examples

SBE requires delivery teams to use conversations to collect meaningful examples that help the team understand the requirements. From examples, tests are created. Good examples make tests better and more business-driven by covering real-world use cases provided by business stakeholders and customers. In the end, tests verify whether the delivery team implemented requirements correctly. You can see the schematics of this process in figure 1.7.

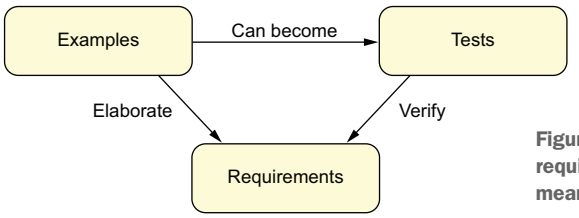

Figure 1.7 Collecting examples that illustrate requirements is the first step to create meaningful, business-driven, automated tests.

Automating ties conversations and examples to system behaviors. Tests return binary responses about every conversation you capture. A conversation either passes your test, meaning the behavior was implemented correctly, or it fails the test, meaning the system is incomplete or broken.

If an example passes the test, you know that the acceptance criterion illustrated by the example is still relevant. If the test is failed, you're notified that the changed code base no longer satisfies the acceptance criteria. If that's the case, the examples should change to reflect that—and sometimes the code has to change, too. (The code could be right and the example now outdated, or the example could be right and the code wrong. In each case, you fix a different thing.)

Why is that? Imagine that the example public transport company introduces new express buses. These vehicles skip most of the bus stops on their way, in order to get to the destination point more quickly. Your team now needs to add express buses to the mobile app. It's a simple change in terms of code: somebody must add a new attribute to the database that determines whether a bus line is an express line or a regular one. Easy peasy. You make the change quickly and then take a lunch break.

That's when all hell breaks loose. (Almost.)

The team forgot that the timetables module isn't the only one affected. The mobile app also features a live map that shows how the buses closest to the user move around. A commuter can check which buses are which in the legend on the map. The legend is generated automatically, but adding the new type of express bus broke the programming logic behind it. As a result, the legend has disappeared. For the few days before you notice the problems, commuters not only aren't able to distinguish express buses from regular buses—they aren't able to find *any* buses on the map.

If you had any documentation in place, the change made it inaccurate and outdated. Nobody updated the document, because your team wanted to have lunch. That's what usually happens: people forget, production hotfixes creep in, the Four Horsemen of the Apocalypse drop by. And when the dust settles, your carefully prepared documentation no longer reflects the current state of the system. In this case, it doesn't tell the reader that there are two types of buses, and it doesn't explain the difference between them. Step by step, with every hotfix and every negligent change, the documentation becomes irrelevant.

1.6.2 Validating frequently

None of these problems would arise if your conversations and examples were automated. When conversations are run as tests, you can regularly track which ones behave correctly and which ones don't. If you test frequently and your specification is exhaustive, you'll get instant feedback after you make a change to the code base.

You can validate during the development process or before a release—what matters is that you must do it often. The more often you test, the sooner you can spot possible errors.

Captured conversations should be validated against both the existing system and new code as it's being written. If you validate conversations frequently, you can have as much trust in the specification as you have in the code. This way, you create a more accessible way to review implemented requirements for all stakeholders.

Because SBE and Gherkin see development as a process of constant discovery through reducing uncertainty about requirements, the model of the system is, by definition, not fully defined from the beginning—it's only defined *well enough*. It evolves continuously based on feedback from stakeholders, and new examples and domain concepts enter the specification as new elements are added to the code. To make sure these new examples fit into the system, delivery teams need a process of *continuous integration*.

> **DEFINITION** *Continuous integration*—A software development practice where members of a team integrate their work frequently. Each team member should integrate as often as possible, leading to multiple integrations per day. Each integration is verified by an automated build to detect integration errors quickly.

If the team uses a testing tool (like Cucumber, a Gherkin-compatible test runner), the tests can be run on each software build. If any errors are signaled, they can be caught early and fixed, letting the "integrate, build, test" process start again—this time, successfully.

1.6.3 Living documentation

As much as we'd like it to be otherwise, only working production code holds the truth about the system. Most specifications become outdated before the project is delivered. Because every product is a machine made out of thousands of moving parts, the dating problem becomes a curse of all software projects.

Outdated documentation may *seem* like a reliable source of knowledge about the system, but it only misleads its readers. An automated, frequently tested specification—as well as the examples included in it—is resistant to such problems. The direct connection between scenarios and code often reduces the damage by cultivating a system of *living documentation*.

DEFINITION *Living documentation*—Documentation that changes along with the system it documents, thanks to the link between the text and the code as well as frequent validation.

When tests keep specifications in check, they let specifications with examples evolve into a documentation system. Using executable specifications as living documentation means taking advantage of automation to facilitate learning within the team and their decision-making abilities. When Gherkin scenarios are free of unnecessary technical bloat, well written, accurate, and full of business-oriented examples and domain vocabulary, they can serve as a *single source of truth* that everyone uses to learn about the functionalities in question.

Thanks to frequent validation, you know that your tests, examples, and conversations are up to date; and when you trust your tests, you can use them as documentation for the entire system. You can track every test back to its origin—the conversation you had with your stakeholders about the requirement. When in doubt, you or anyone else on your team can always check the captured conversation. Frequent validation also guarantees that the documentation must change every time the underlying code changes, because the documentation is connected to the code through tests.

A living documentation system should benefit everyone. Specifying collaboratively, illustrating requirements with examples, and refining specifications for readability—all these measures should involve everyone who matters in the requirement-analysis process, or a few dedicated people can make the requirements as easy to understand as possible for everyone else. Everyone involved should be able to read the results, too. Tools such as Relish, Cucumber Pro, and CukeHub can even integrate with a code repository of your choice and publish the scenarios in a private cloud where you can collaborate and share executable specifications and test results with other team members, as easily as you can share a document in Google Docs.

1.7 Capturing conversations as executable specifications

Okay, so automating conversations offers a lot of benefits. But *how* do we automate them? At the beginning of section 1.6, I promised that we'd come back to the topic of recording conversations in a language that will help you optimize them for automation. This section discusses the refinement process that makes free-flowing conversations easy to automate (see figure 1.8).

Specification workshops allow for having conversations. Programmers and testers are responsible for automation. How does the translation process happen? Should programmers store conversations as comments in their testing code? That would be ridiculous—but the records have to be written somewhere, don't they? A free-flowing conversation is, by definition, an unreliable medium that only stimulates short-term memory. We need long-term storage.

As introduced earlier, Gherkin is *the* tool for capturing conversations about requirements in a formalized way, clarified by extracting essential information and removing noise. Gherkin facilitates knowledge sharing among all stakeholders,

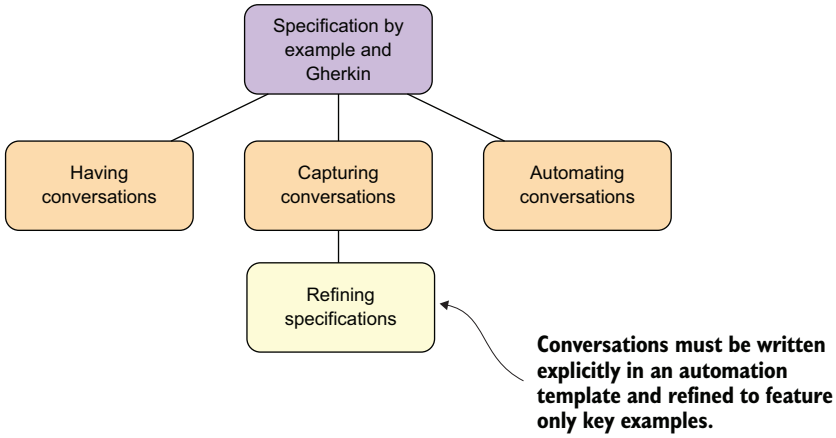

Figure 1.8 The capture process aims to preserve verbal product-design chats and translate them to lightweight, long-term, formalized stories that, in time, can be used to put together system documentation.

regardless of their technical skill. It does so by conveying tests and requirements in a ready-for-automation template that's expressed in plain English and that uses the ubiquitous language of a product.

Gherkin focuses on capturing conversations as *scenarios*. Scenarios preserve essential information and remove noise by extracting concrete actions from conversations.

DEFINITION *Scenario*—A concrete example that illustrates a business rule

Following is an example of a scenario. Remember the conversation about how the mobile app for journey planning should work?

> *"Let's imagine that you're going to the city on a business trip. When do you get there? … If you're a bit happy-go-lucky and arrive in the city an hour before the meeting, you don't have enough time to check where you are."*
>
> *"Wow, we might need to implement a GPS geolocation functionality, so we could help users know their current location."*

Here's the same conversation expressed in Gherkin:

```
Given a commuter that enabled mobile location tracking
 When the commuter wants to plan a journey
 Then the starting point should be set to current location
```

This sequence is called the *Given-When-Then* template. I'll talk about it in detail in chapter 2, where you'll learn the basics of using the template.

Thanks to its focus on user actions, Gherkin is a great language for conveying *behavioral requirements*. Just as having conversations improves a delivery team's short-term understanding, capturing conversations ensures that they don't let that knowledge slip through their fingers in the future. Scenarios achieve that and remind us

that we don't need 100-page functional requirements documents to capture what's valuable. We don't even have to write all the scenarios up front. We can capture a few scenarios at a time, as we discuss each new requirement. A few months in, we'll have a huge library of relevant scenarios. We only need to be consistent.

> **DEFINITION** *Behavioral requirement*—A requirement formed as a story about how users behave when they interact with the system. Whereas normally requirements can be formed as abstract statements, behavioral requirements always talk about examples of using the system.

The contents of the template should use nontechnical language that relies heavily on real-world business concepts. Notice how the example mentions *commuter*, *journey*, and the *starting point*—concepts borrowed from the business vocabulary of the public transport company—but doesn't say anything about low-level development procedures or the application's user interface. Scenarios captured using the Given-When-Then template should stay at a business-readable, code-free level at all times, improving the domain model and building its ubiquitous language.

> **WARNING** If you see anything about a connection to the database in a Gherkin scenario, or read about buttons or any other UI element, somebody made a huge mistake.

Because programmers and testers can automate anything put in the Given-When-Then template, scenarios written in Gherkin become *executable specifications*. This book will teach you to write executable specifications in Gherkin and use the Given-When-Then template. You'll also learn the rest of the Gherkin syntax required to capture design conversations in a form that easily translates to executable specifications.

> **DEFINITION** *Executable specification*—A specification that can be run as an automated test

The syntax serves as a link between speech, text, and automated code. It lets you progress naturally from one to another. Gherkin also provides techniques to organize scenarios into full documents, link similar behaviors, and simplify capture and automation, all while keeping things at a business-readable level derived from the ubiquitous language.

Most executable specifications contain many scenarios, and every scenario needs multiple examples. In its rough form, an example is like a quick note or a doodle. It makes sense when you look at it a day after you made it, but try examining it six months later—not so meaningful anymore, right? That's why successful teams don't use raw examples; they *refine* them. A team extracts the essence of key examples and turns it into clear, unambiguous, organized specification documents, as shown in figure 1.9.

As new requirements appear, acceptance criteria generate new examples, and every example generates a new scenario. To refine executable specifications, teams merge similar examples, reject examples that introduce noise, and choose the most

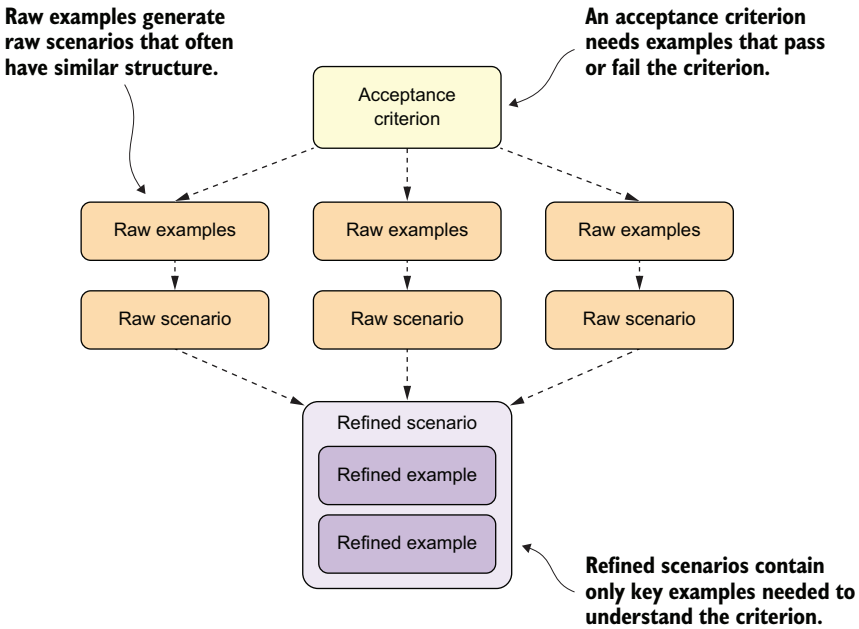

Figure 1.9 The process of refining raw examples extracted from collected acceptance criteria into refined scenarios with key examples

meaningful or descriptive examples. The result is an executable specification in its final form, ready to become the foundation for the living documentation system.

1.8 Making software that matters

You've now begun the journey of mastering executable specifications written in Gherkin according to SBE's key practices. As you learn more about SBE and Gherkin, we'll focus on practicing techniques that help you avoid common specifying pitfalls. When software engineers and designers don't put enough thought into their specifications, the cost is measured in weeks of work and hundreds of thousands of dollars wasted.

The benefits of SBE and Gherkin go far beyond reducing rework. You'll get better insight into your business domain and reduce friction caused by inevitable translation costs that come up when a business requirement becomes working software. People made these tools and processes because they wanted to guarantee that the software they help build will make sense to customers. They wanted to make software that matters.

SBE, BDD, or ATDD?

When I started my journey with executable specifications, like many other practitioners I was confused by the naming issues around the topic of agile acceptance testing. When I found out that many people call SBE *behavior-driven development* (BDD) or *acceptance test-driven development* (ATDD), I didn't understand the difference.

(continued)

My confusion was deepened by the fact that I became interested in SBE after reading Gojko's book, but the first project where I was able to practice writing executable specifications used Gherkin. In his book, Gojko wrote that he didn't "want to use any of the *Driven Development* names, especially not Behavior-Driven Development." But Gherkin was invented by Dan North and Chris Matts, and Dan North is the main face of BDD. I was perplexed.

I wanted to avoid naming controversies, because they aren't key to what you're going to learn. I honestly admit that I borrowed freely from both fields, trying to create a mix that will maximize benefits and minimize mental load. Dan North calls BDD a *methodology*; but, quoting Gojko, what I wrote here doesn't form a fully fledged software development methodology. My only goal is to teach you to write great Gherkin specifications *using* SBE practices. So whenever I talk about a practice or an idea derived from SBE, I'll tell you that up front. Everything else will appear under the umbrella term of *Gherkin* and *good Gherkin practices*; if you want to read more about it, you can assume it comes from the field of BDD.

Because this is a book about practical application of executable specifications with examples, it mainly deals with capturing conversations in Gherkin and refining examples. It's a long-ignored topic due to Gherkin's seemingly easy syntax and elusively low entry barrier. Many software engineers and designers think they need a quick tutorial and then can start writing. It's only a simple Given-When-Then sequence, right?

Yes and no. Everything depends on the project you're dealing with. At first, having executable specifications will yield better alignment without much training—but complex products with complicated business domains can go astray quickly. Hundreds of requirements will produce hundreds of Gherkin files you have to manage. And every file will contain multiple scenarios, and every scenario will attach additional example. That sounds like Gherkin and SBE don't fit huge projects well; but, truth be told, huge projects will stretch every process and tool. As you'll see, executable specifications with examples shine the brightest in complex environments—but that's why I'm writing a book, and not a blog post.

You don't have to be able to write testing code to read the book. I'll cover automating conversations only as long as it introduces good patterns that will make life easier for your engineers. Having said that, we should always value business-oriented specs over specs that are easy to automate. Similarly, I won't talk about anything related to having product-design conversations during specification workshops, unless it directly impacts you when writing specifications in Gherkin. There are other resources that teach these skills well enough.

I do expect you, however, to understand the basics of the automated testing *process* and why it matters. Practical knowledge about the QA process will be helpful in some of the later chapters. If you have a technical background or are experienced in working with developers and QA engineers in any agile methodology, you'll be fine. I also

expect you to understand what it takes to release a product, from its conception through the public announcement to long-term maintenance. Some of the things we'll discuss will cover not only initial requirements but also possible changes in scope that a product can face at a later stage of its life cycle.

What will you learn? The next chapter explores Gherkin and SBE in practice. You'll begin by capturing requirements and acceptance criteria as executable test cases. As you progress through the book, you'll tackle more-advanced topics. You'll learn to write good scenarios. You'll see how to choose good examples. You'll design business-oriented error checks. When the time comes, you'll move on from thinking about suites of specifications, and you'll learn to organize scenarios into groups of coherent specification documents that readers can navigate easily. I'll also talk about how the ubiquitous language shapes examples and scenarios, and how to evolve specifications into a living documentation system over time.

Right now, though—right now, welcome to specification by example and Gherkin.

1.9 Summary

- A specification is a description of the system design required to implement the system.
- Acceptance criteria let you review whether you've built a complete system.
- A ubiquitous language is a common language among developers, business stakeholders, and end users. It makes every stakeholder sure they're talking about the same things.
- Specification by example is a business-analysis process aiming for "just enough," just-in-time software design. Lightweight examples provide enough initial context to start development and are later refined into more-sophisticated forms.
- Gherkin is a business-readable language for writing specification documents. Gherkin's practitioners capture conversations about requirements in the form of behaviors—also called scenarios—which are examples of how the system is expected to behave.
- An executable specification is a conversation captured using the Given-When-Then template with a corresponding acceptance test. The acceptance test makes sure the delivery team has implemented the underlying requirement correctly.
- Every executable specification's life cycle starts with a specification that later becomes an automated test. Automating the specification ensures that it stays up to date, because the captured conversation is directly tied to testing code. This way, tests become documentation.
- Gherkin and SBE arm you with software development techniques that facilitate knowledge sharing, reduce short-term waste without sacrificing long-term documentation, and help the delivery team deliver software faster and without rework thanks to meaningful, concrete examples of system behaviors that ensure everyone's on the same page.

SOFTWARE ENGINEERING

"Does a great job taking concepts from good architectural practices and applying them to the world of collaborative specifications."

—From the Foreword by Gojko Adžić, author of *Specification by Example*

"The missing manual for writing great specifications. I wish this book had existed five years ago!"

—Craig Smith, Unbound DNA

"Will jolt you into best practices, give you fresh perspectives, and reinvigorate your commitment to this business-critical skill."

—Dane Balia, Hetzner

"The complete book on how to write great specifications. Most of us know bits and pieces, but to truly grok it, you need this excellent guide."

—Kumar Unnikrishnan
Thomson Reuters

Writing Great Specifications

Kamil Nicieja

The clearest way to communicate a software specification is to provide examples of how it should work. Turning these story-based descriptions into a well-organized dev plan is another matter. Gherkin is a human-friendly, jargon-free language for documenting a suite of examples as an executable specification. It fosters efficient collaboration between business and dev teams, and it's an excellent foundation for the specification by example (SBE) process.

Writing Great Specifications teaches you how to capture executable software designs in Gherkin following the SBE method. Written for both developers and non-technical team members, this practical book starts with collecting individual feature stories and organizing them into a full, testable spec. You'll learn to choose the best scenarios, write them in a way that anyone can understand, and ensure they can be easily updated by anyone.

What's Inside

- Reading and writing Gherkin
- Designing story-based test cases
- Team collaboration
- Managing a suite of Gherkin documents

Primarily written for developers and architects, this book is accessible to any member of a software design team.

Kamil Nicieja is a seasoned engineer, architect, and project manager with deep expertise in Gherkin and SBE.

To download their free eBook in PDF, ePub, and Kindle formats, owners of this book should visit
www.manning.com/books/writing-great-specifications

ISBN-13: 978-1-61729-410-5
ISBN-10: 1-61729-410-1
54499
9 781617 294105

MANNING $44.99 / Can $59.99 [INCLUDING eBOOK]