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 6

brief contents

6 The life cycle of executable specifications

This chapter covers

- Working with an executable specification throughout its life cycle
- Understanding requirements' precision level
- Using examples in different development phases
- Understanding what happens after implementation

In its original meaning in systems theory, *feedback* is the exchange of data about how one part of a system is working—with the understanding that one part affects all others in the system—so that if any part heads off course, it can be changed for the better. In SBE, a specification suite is the system, every executable specification is a part of that system, and the delivery team is the recipient of the feedback.

As anyone who works in a corporate environment knows, there are two kinds of feedback:

- *Supporting* that lets people know they're doing their job well
- *Critiquing* that's meant to correct the current course of action

The same rules apply to the systems of feedback in SBE. Back in 2003, Brian Marick wrote a series of blog posts about the concept of an agile testing matrix.[1] The series is also one of the oldest articles about modern testing methods I know of that suggested dropping the name *tests* and replacing it with *examples* for business-facing testing.

Marick organized his quadrant into distinct quadrants with two axes (see figure 6.1). The vertical axis splits the matrix into business-facing tests, such as prototypes and exploratory testing, and technology-facing tests, such as unit tests and performance tests. Unit tests deal with low-level testing; they make sure small components of code run well. Performance tests determine how a system performs in terms of responsiveness and stability under workload.

Our area of interest lies in the upper half of the matrix, though: it's split into tests that *support the team's progress* and tests that *critique the product*. Tests that support the team help the team write new code before they have a working product. Tests that critique the product look at a finished product with the intent of discovering inadequacies. As you can see, this is similar to the feedback system that powers any human organization.

The question we'll explore throughout this chapter is when examples should support the team's work and when they should critique the product. You'll see how examples and automation can give delivery teams feedback of both kinds. And you'll come to understand how feedback forces at the core of every executable specification suite work much like a development manager: constantly evaluating the job done by

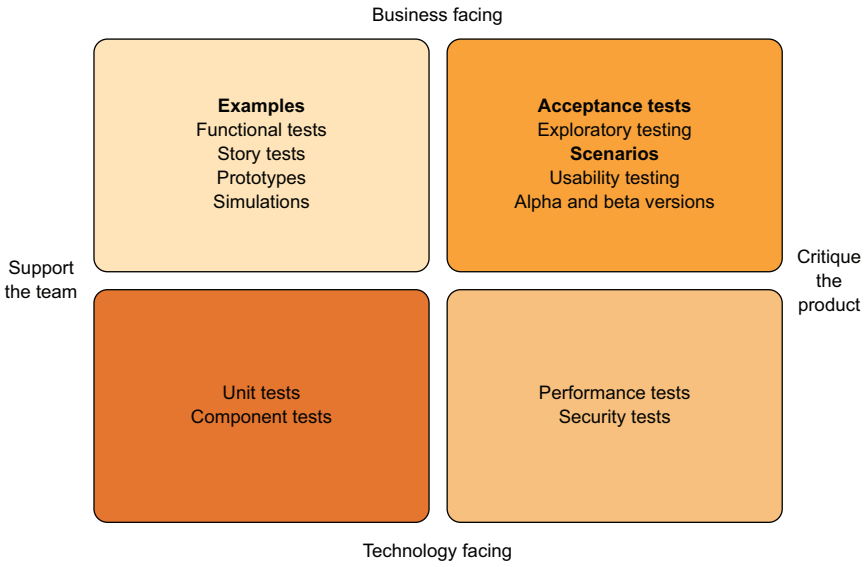


Figure 6.1 The agile testing quadrants. I've highlighted the tests that will be important in this analysis.

[1] See Brian Marick, "My Agile Testing Project," on his blog, "Exploration Through Example," August 21, 2003, http://mng.bz/TkO1.

programmers, analysts, designers, and testers; praising the team for their successes; and pointing out failures.

To see how feedback loops in SBE help uncover contradictions and uncertainty that can hide even in well-analyzed requirements, I'll also talk about the life cycle of an executable specification. In product design, *life cycle* can be defined as the process of bringing a new product or a new feature to market and then maintaining it over time.

Chapter 1 said that all software development processes follow similar phases as a functionality progresses from conception to release, such as planning, building, releasing, and getting feedback. These phases are also a called the *software development life cycle*. I also said that, traditionally, specifications belong to the planning phase because that's when delivery teams first interact with new requirements (as presented in figure 6.2).

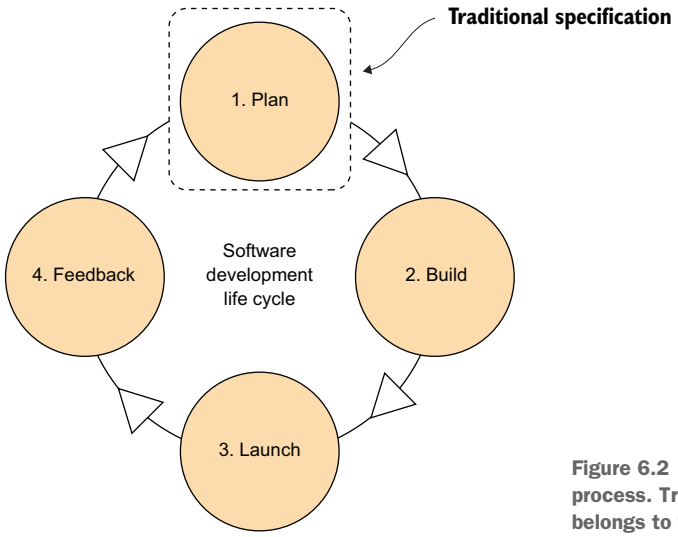


Figure 6.2 The software development process. Traditional specification belongs to the analysis phase.

SBE is different because it doesn't see specification as a singular phase but rather as a process that spreads across multiple phases as requirements evolve and change (see figure 6.3). Specifications become tests; and tests, as you'll see in chapter 7, become documentation.

By *evolving requirements*, I mean that the precision level of your requirements changes throughout the life cycle of an executable specification. Every requirement starts as a vague goal with a high level of uncertainty. As you implement it, uncertainty decreases, because the requirement gradually becomes working code. As you'll see, SBE and Gherkin have a different feedback loop at each stage of the life cycle, which helps decrease uncertainty even further than in other software development processes.

To understand this process thoroughly, we'll track how a raw requirement *becomes* a Gherkin specification—from a broad concept to a working implementation. This time

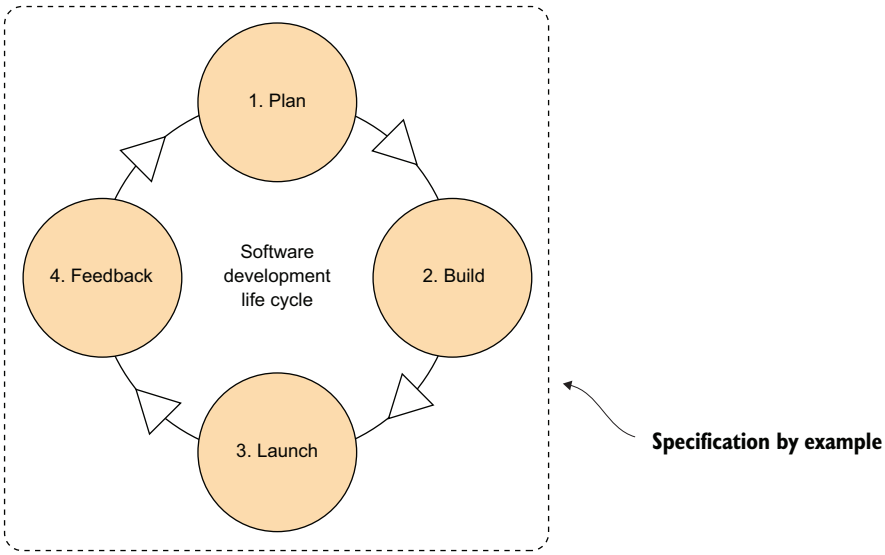


Figure 6.3 SBE in the software development process. Specification spreads throughout multiple phases.

around, you'll specify a new feature in a simple mapping application called Mapper. Maps are by definition instruments of changing precision. You can use one to look at continents and countries from a high-level view, but you can also buy a local map of your city and use it to find your home. A map is a great metaphor: just as there are low-resolution and high-resolution maps, there are low-resolution requirements and high-resolution executable specifications.

6.1 End-to-end overview of the process

In this and the next section, we'll look at a full overview of what *life cycle* and *precision level* mean. Every software feature goes through several development phases such as analysis, design, implementation, testing, and maintenance. This is what a life cycle is. As discussed in chapter 2, Gherkin specifications are also called *features*, and they go through similar phases.

At its core, every executable specification is the result of a five-step process (see figure 6.4):

1. Understanding business goals
2. Analyzing the scope of requirements through examples
3. Designing the solution by deriving scenarios from acceptance criteria and examples
4. Refining scenarios until you can implement the behaviors from the specification
5. Iterating the specification over time

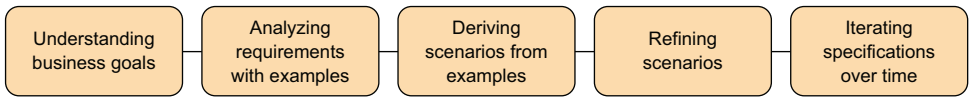


Figure 6.4 The life cycle of executable specifications

As you cross each threshold, the precision level of your analysis increases (see figure 6.5). The more precise you are, the better you understand the implications of implementing a requirement.

Business goals are broad directives that sometimes don't provide a specific solution. Requirements are less abstract; they define what needs to be built for whom in order to achieve the high-level goal. Solutions are precise plans for features, interaction flows, and user interfaces. Because software is comprised of thousands of moving parts, no solution is fully precise until it's implemented as working code. And only after the code is released to production do you get the feedback needed to asses whether a business goal has been met, which removes any remaining ambiguity.

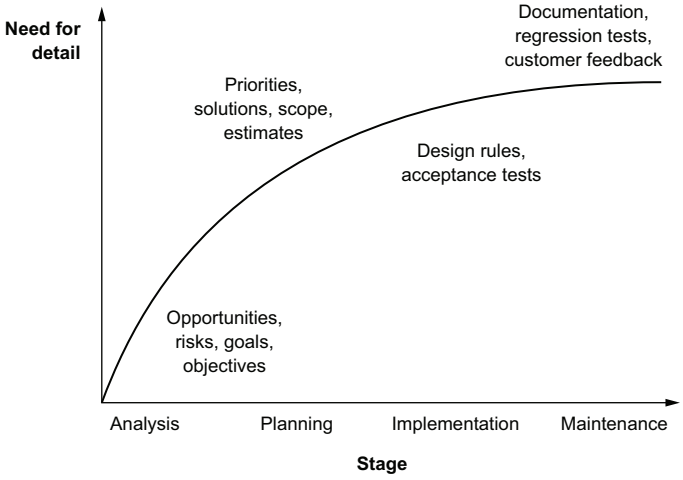


Figure 6.5 As the need for detail increases, so does the precision of artifacts that appear during development.

6.2 Understanding business goals

To see how the precision level increases throughout the life cycle of an executable specification, we need to start with the least specific phase: discussing business goals (see figure 6.6). For this example, assume on the Mapper project, your responsibility is to lead a team of developers, designers, and testers. Management is looking for a way to increase Mapper's presence among small businesses and get smaller enterprises to pay to be featured on your online maps. Right now, your maps feature only a few of the most popular outlets of international companies located at prominent locations, but management would like to include more firms. Your team has been tasked with making that change happen.

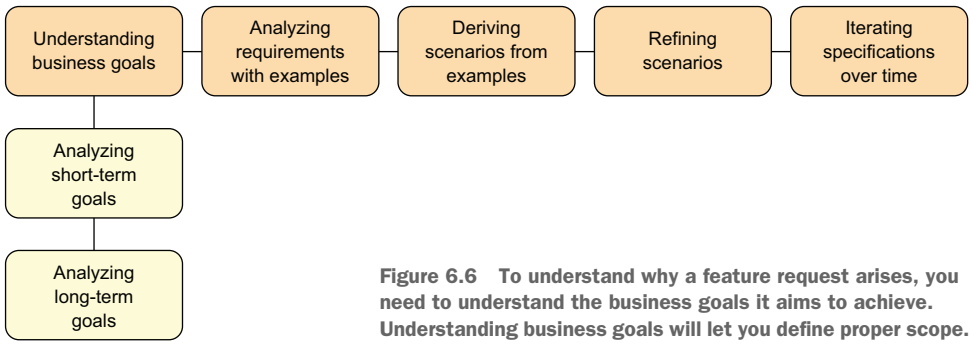


Figure 6.6 To understand why a feature request arises, you need to understand the business goals it aims to achieve. Understanding business goals will let you define proper scope.

TIP When you're trying to elicit requirements, raising the discussion to the level of business goals lets you deal with scope and priorities more efficiently. As you raise the discussion to the level of goals, the number of things you have to talk about decreases. That makes it easier to focus on the essentials.

You and management agree on actionable goals that the team should aim to meet in the next six months; see table 6.1. You have to understand both the short-term and the long-term contexts. If you knew only the short-term goal, you'd devise a temporary solution. For example, your team could add new businesses to the platform manually and still achieve the goal. Because the absolute number of businesses featured on Mapper is low at the moment, doubling that number would take only a few minutes of work.

Table 6.1 High-level goals for the next six months

Perspective	Goal
Short-term	2x increase in businesses featured on Mapper's platform
Long-term	Establish presence in the unsaturated segment of small businesses in order to seek growth

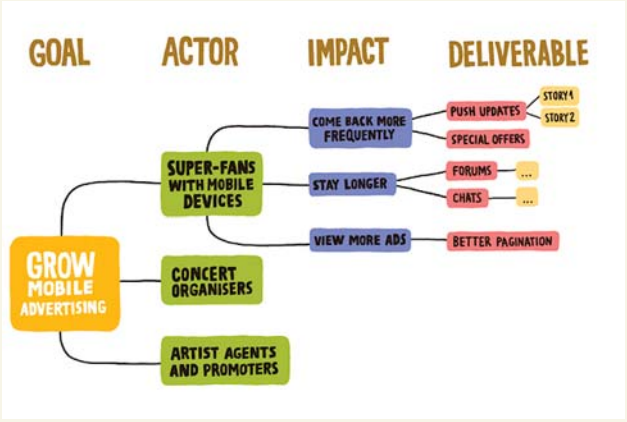
Only when you also look at the long-term goal you can see that the company is looking for a solution that will let it achieve sustainable growth in a new market segment. Meeting such a goal will require you to take different actions.

Other resources for strategic planning

Impact mapping is one of the best methods I know for creating medium-term strategic plans. An *impact map* is a visualization of scope and underlying assumptions, created collaboratively by senior technical and business staff. It's a mind map that's grown during a discussion that considers the following four aspects:

- *Goal*—Why are we doing this?
- *Actors*—Who can produce the desired effect? Who can obstruct it? Who are the consumers or users of our product? Who will be impacted by it?
- *Impacts*—How should our actors' behavior change? How can they help us to achieve the goal? How can they obstruct us or prevent us from succeeding?
- *Deliverables*—What can we do, as an organization or a delivery team, to support the required impacts?

I like impact maps because they make assumptions hierarchical and testable. What does that mean? Let's say you have an impact map with a goal to grow mobile advertising for a website.



Example of an impact map from www.impactmapping.org/drawing.html

First, you define the success metrics for your goal. Then, you identify several actors who can help reach you that goal, including super-fans with mobile devices. One possible impact could be that they stay longer on your website and increase advertising exposure. You figure that features such as forums and chats, among others, might help achieve that impact.

Let's say that after implementing several deliverables, you realize that you were wrong in thinking that forums and chats would increase engagement. You may have several other, similar deliverables on your map, but now you know that this branch of the map isn't as impactful as you thought it would be—so you might discard that impact or even discard that actor. If you did the latter, all the other deliverables for super-fans with mobile devices would be automatically discarded as well, because the map is hierarchical. Thus you can treat the map as an easy-to-maintain, visual, testable product backlog that keeps changing its shape as your knowledge about the world grows.

You can read more about impact maps in *Impact Mapping* by Gojko Adžić (Provoking Thoughts, 2012).

6.3 Analyzing requirements with examples

As you may recall from chapter 1, an SBE process starts with deriving scope from goals. The previous section provided you with a goal. Now you need the scope. There are several methods to derive scope (see figure 6.7). In recent years, user stories have risen to become the most popular method for defining and discussing scope among agile teams; this section will guide you through the process of creating user stories, illustrated with examples.

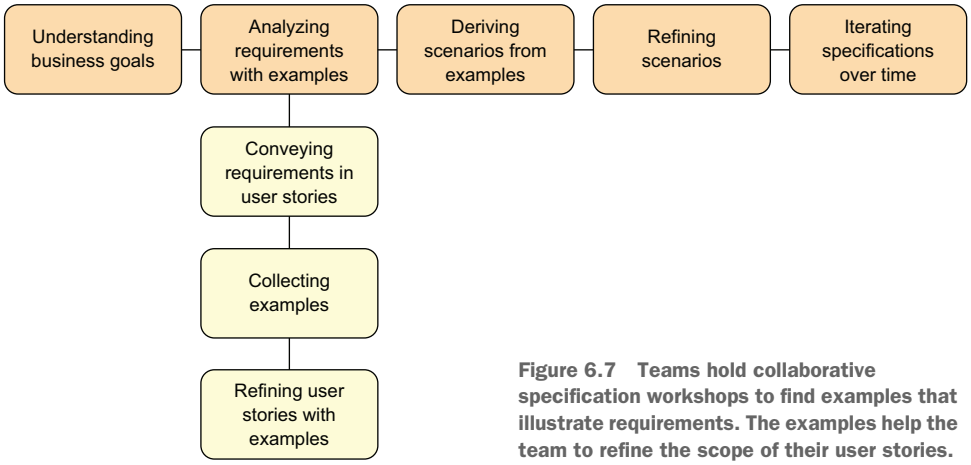


Figure 6.7 Teams hold collaborative specification workshops to find examples that illustrate requirements. The examples help the team to refine the scope of their user stories.

6.3.1 Conveying requirements as user stories

A user story lets you convey a glimpse of a requirement as a product backlog item. The item becomes a *ticket for a future conversation*. Through user stories, you can negotiate the priority of any given requirement, discuss possible solutions and scope, and make some estimates. They're used for *planning*.

The previous section talked about short- and long-term business goals. Even if you acknowledge both contexts that are important to your company, that's only one side of the story. Customers don't care about the goals of businesses; they want businesses to bring them value. Customer-oriented firms know that and use user-centered tools to align their strategies to their customers' interests.

That's what user stories are for. Writing a user story lets you restate your business objective as a customer benefit.

Listing 6.1 [OK] Your first user story

```
In order to let new customers discover my company
As an owner of a small business
I want to add my company to Mapper's platform
```

TIP I talk more about user stories in chapter 8. You'll learn alternative formats to express your stories so that you'll be able to aim for understanding rather than strict conformance to any single format. Stay tuned! Right now, you only have to focus on the three key elements that all user stories share—*who*, *what*, and *why*.

Thanks to your knowledge about your market, you know that making Mapper a new distribution channel will encourage small businesses to add themselves to the platform. They will essentially do the job for you, if you help them find new clients.

User stories are crucial to increasing precision to the level an executable specification needs. But stories and scenarios are separate creatures, as shown in table 6.2. User stories have acceptance criteria. Executable specifications have acceptance tests. Without the criteria, there can be no tests. A delivery team derives new executable specifications and new scenarios from user stories.

Table 6.2 A comparison of user stories and executable specifications

User story	Executable specification
Discarded after implementation	Kept after implementation
A unit of change	An effect of the change
Has acceptance criteria	Is an acceptance test
Produces short-term results, such as cards or tasks	Produces long-term, living documentation

At this stage, you already suspect that your team will have to build some kind of a form that will allow companies to sign up and mark themselves on your maps. You know that because analyzing business goals and writing the user story increased the precision level of the requirement to the point that you can begin to devise a specific *solution*.

Placing the responsibility for a solution on the development team is a great way to obtain the right scope for a goal. The executive team may already have ideas about the solution, derived from their intuitions and expertise; but they made you the product owner, so you're the decision maker.

When a requirement or a business objective contains implementation details, it usually means somebody's trying to slip in a predetermined solution, binding it unnecessarily with the problem your organization is trying to solve. It could be a team member, a manager, someone from marketing and sales, or even a bossy customer.

Henry Ford famously said, "If I had asked people what they wanted, they would have said faster horses." Visionaries use this quote as a beaten-to-death excuse for ignoring customer feedback. I think customers clearly told Ford what they wanted: they told him that speed is the key requirement for transport. But because they weren't engineers, they weren't able to say that cars would satisfy the requirement.

People will always use solutions to help themselves imagine consequences of any given requirement, because it's a natural way of thinking. But as someone who works

with technology, you should strive to extract unbiased, pure requirements from their solutions. Only when you decide what will work best for the company should you move on to writing down an executable specification.

Storing user stories in a product backlog

A user story is only a token for a future conversation. It's a reminder that when the right time comes, you'll have to discuss the such-and-such requirement with your stakeholders in order to implement it correctly.

Even though user stories shouldn't by any measure *replace* conversations, you can prepare notes in advance that will help you get up to speed after you pull a story out of the product backlog. (A backlog is a lot like a freezer: some stories don't age well.) You don't want to be overly specific, of course. Specificity at this stage could constrain your flexibility in the future. You should never treat user stories as a to-do list. They're more like a list of guidelines: directions that you suspect you might explore in the future. But stories can stay in the freezer for months, so you may want to provide *some* details to remind you of its purpose back when you put that particular story in the backlog.

To specify my stories, I use a four-element template based on a simplified story-elements template first shown to me during a workshop by David Evans, who is a veteran of agile testing and an active member of the agile community. For each story, I write down the following:

- *Stakeholders and their interests*—Along with the primary actor who has the most interest in the story and is featured in the story, I sometimes list other stakeholders who could be affected by the story.
- *A trigger*—The event that causes the new behavior to be initiated or invoked by a user or by the system itself.
- *The main success scenario*—Intentions and outcomes that should guarantee the primary actor's success (remember not to over-specify UI or implementation details!).
- *Acceptance criteria*—Two to five one-line descriptions that sufficiently identify each testing condition for the story to be verified.

I put these notes in the description of the story in the backlog.

You can see that each of the four notes will help me prepare Gherkin scenarios more quickly after I pull the user story from the backlog. I'll already know the primary actor. From the trigger, I'll have some idea about the `Given`s. The main success scenario will help me imagine the `When`s and `Then`s. And from the list of acceptance criteria, I'll be able to estimate how many scenarios an executable specification for this user story will have.

Sometimes notes don't age well. Requirements can change over time. But notes can still help you in such situations: you can always compare your past notes with your current direction. The difference between the former and the latter will be the sum of the learning your team has accomplished during the freezer time.

The template can also be useful when you gain new insights about the story, but it's still not the right time to implement that story—for example, when you gather new customer feedback, but have other priorities at the moment. Updating the template can preserve your new insights for the future discussion.

6.3.2 Collecting examples

You now have a user story that you can put in your team's backlog. You also have a rough sense of the amount of work you'll face: you know what's expected of you, and you've shared the news with the engineers, who told you their first impressions and initial ideas of what could be done to meet the requirements. What happens next?

Chapter 1 taught you that after you derive scope from goals, you should start *specifying collaboratively* to *illustrate requirements with examples*. Specifying collaboratively means domain experts, product managers, developers, testers, and designers working together to explore and discover examples that will become Gherkin scenarios. Let's assume that in the Mapper example scenario, you decide to organize a workshop.

TIP It's your job to be a facilitator and to extract relevant information from whatever is said. Starting by asking for rules is okay as long as you don't expect the rules to be a fully developed list of acceptance criteria. They'll be messy and, probably, contradictory or inconsistent. As a technologist, you should constantly challenge and refine the requirements your team takes on.

During the workshop, your team comes up with examples of small businesses based in your town that would be most likely to join Mapper's platform (see table 6.3). You also add a few ideas of how particular businesses may use Mapper to lead new customers through their conversion channels.

Table 6.3 Examples of relevant small businesses

Business name	Business type	Features for lead generation
Deep Lemon	Restaurant	Showing customers business hours
The Pace Gallery	Art gallery	Advertising expositions
French Quarter Inn	Hotel	Booking rooms
Green Pencil	Bistro	Showing customers business hours
Radio Music Hall	Concert hall	Advertising concert programs
City Cinema	Movies	Showcasing new films and ticket prices
Christie's	Pub	Showing customers business hours

What a diverse bunch of businesses! Having them sign up to Mapper would lead to a lot of healthy growth. The difficult part is that they would require different features in order to find the platform valuable, and some of those features would be as complex as integrating with external booking systems and payment processors. But for now, Mapper's customers will have to deal with the limited scope that we'll implement in this chapter.

At the analysis stage, delivery teams use examples to understand the business context of their requirements—just as you did when you illustrated your business goals with examples of use cases that might help you achieve those goals. Delivery teams can also use examples to check whether the designers, programmers, and testers are in sync with the domain experts—again, just as you did during your specification workshop. This is clearly the *supporting role of examples* that I talked about when I introduced the agile testing matrix in figure 6.1. Is there a relationship between the two critical diagrams in this chapter—the matrix from figure 6.1 and the life cycle diagram from figure 6.5? As you can see in figure 6.8, you could easily rework the life cycle diagram to show the supporting role of examples in the early stages of development.

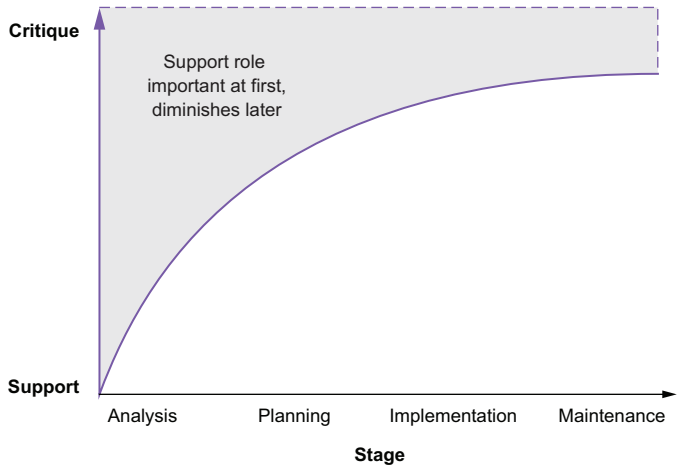


Figure 6.8 Before they're automated, examples support delivery teams in understanding the scope of requirements—which is important at the beginning of the life cycle.

The supporting role of early examples means delivery teams can use them to *support writing new code* when they work on new functionalities. In a way, the right examples *provoke* the right code, guiding the development process.

But as the requirements become more precise, the supporting role becomes less important than the critiquing role (see figure 6.9). By *critiquing*, I mean that examples begin to *challenge the requirement*. For example, what if the examples collected by the delivery team are wrong? Some surely will be.

Examples at the critiquing stage usually have something they didn't have earlier: an actual iteration of working software. When working software becomes available, it's

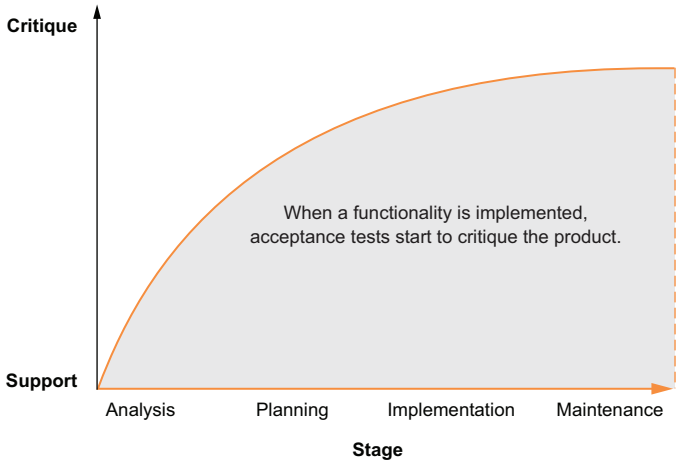


Figure 6.9 As requirements get more precise, examples are challenged by the working product.

easier to verify whether your initial analysis was right. Sometimes, the business expert will forget things that real users will need. They may also be misguided or may champion a preferred solution due to a personal agenda. And sometimes, you can end up overengineering the solution and worrying about too many examples that aren't useful in the real world.

6.3.3 Refining user stories with examples

Let's assume that building a feature for so many kinds of businesses proved to be too difficult for Mapper. Having a deadline to meet and limited resources you can use, you decide to reduce the scope. To choose a customer segment that will allow you to easily expand to other businesses in the future, you must look for a *carryover customer*.

> **DEFINITION** *Carryover customer*—An example of a real customer who shares behavioral traits with as many other customers as possible and becomes a model for an average consumer. If you design a product or a feature for the right carryover customer, other segments of the market should find it valuable, too.

Did you notice that all of the gastronomy businesses in table 6.3 share the same feature—showing customers business hours—that would drive their conversion rates? Other businesses, such as shops, pubs, and clubs, share this trait, too. After some discussion, the Mapper team agree that restaurants and bistros make good carryover customers.

Listing 6.2 [BETTER] Refined user story

```
In order to let my customer know where and when they can come in
As an owner of a gastronomy business
I want to add my company to Mapper's platform
```

You believe that implementing this user story will fulfill the short-term business goal you were tasked with. While you test the waters with gastronomy, the user stories for other businesses will wait patiently in the backlog for their turn.

> **TIP** Features can usually be split only based on *technology*. This is a limiting approach. For example, if you wanted to split a report-generating feature into smaller portions, your first instinct would probably be to split it based on its *technological* ability to generate reports in different formats such as .pdf, .csv, and .xls. User stories and requirements, on the other hand, can be split by *value*. In your analysis of Mapper's platform, you choose the most valuable customer segment and split a small capability that would bring this segment a lot of value without a lot of effort on your side.

You wouldn't be able to make a confident decision without the validation provided by the examples you collected. That's their power.

Collecting, analyzing, and refining examples creates a powerful feedback loop within any project that uses SBE. Examples helped you define the scope of the features and split user stories into smaller, more precise backlog items. Even though you're yet to see an executable specification that your Mapper team could use, it should already be clear why examples lie at Gherkin's center.

Eliciting better requirements with Feature Injection

The process you've been using throughout the chapter to elicit requirements is similar in design to *feature injection*: a technique that iteratively derives scope from goals through high-level examples.[a] In feature injection, teams first "hunt for value" by creating a model for delivering business value and listing underlying assumptions instead of trying to describe the value with simple numbers such as revenue goals. You created a simple model for value delivery when we discussed Mapper's short-term and long-term goals, why the goals are important, how they influence Mapper's business model, and which stakeholders demand that you achieve the metrics.

Once a model is defined, you do more than just evaluate whether to accept or reject a suggested feature. You can proactively create a list of features that drive toward delivery of business value based on your models. You did that when you wrote your first user story and refined it to better fit the value model.

Injecting features provides a set of *happy paths* to create the outputs that will deliver the business value. But doing so doesn't provide all possible variations of input that can occur and that may affect the outputs, or all cases that need to be considered for successful delivery. You may recall happy paths from section 5.3.3, where I talked about exploratory outcomes. *Exploratory outcomes* pursue possible unhappy paths, leading to new testing ideas. When new examples are generated, you can put them together in an executable specification—which is what you're about to do for Mapper.

[a] Feature injection was created by Chris Matts and then expanded with Rohit Darji, Andy Pols, Sanela Hodzic, and David Anderson over the years 2003–2011. For more information, see "Feature Injection: Three Steps to Success" by Chris Matts and Gojko Adžić, *InfoQ*, December 14, 2011, http://mng.bz/E5fS.

6.4 Deriving scenarios from examples

In this section, you'll step over another precision threshold as the user story you chose (listing 6.2) finally transforms into a draft of an executable specification. To do so, you'll finally write your scenarios (see figure 6.10).

TIP A story becomes an executable specification when you're sure it's worth investing time and effort in it.

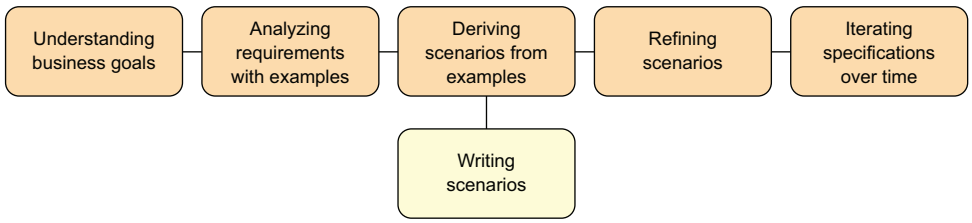


Figure 6.10 Examples serve as a basis for all scenarios to come. Over time, the team should optimize scenarios for readability and remove confusing, redundant examples.

From the examples you collected, you derive a list of acceptance criteria for the user story:

- Every new business should provide a name and a location to display on the map.
- Every business should provide business hours for each day of the week.

Let's take the relevant examples of pubs, restaurants, and bistros from the previous section to write the first draft of an executable specification.

Listing 6.3 [OK] First draft of an executable specification

```
Feature: New businesses

  Scenario Outline: Businesses should provide required data

    Given a restaurant <business> on <location>
     When <business> signs up to Mapper
     Then it should be added to the platform
      And its name should appear on the map at <location>

    Examples:
      | business      | location                        |
      | Deep Lemon    | 6750 South Street, Reno         |
      | Matt's        | 9593 Riverside Drive, St. Louis |
      | Back to Black | 8114 2nd Street, Stockton       |
      | Green Pencil  | 8583 Williams Street, Glendale  |
      | Le Chef       | 3318 Summit Avenue, Tampa       |
      | Paris         | 2105 Briarwood Court, Fresno    |
      | Christie's    | 714 Beechwood Drive, Boston     |
      | The Monument  | 77 Chapel Street, Pittsburgh    |
      | Anchor        | 110 Cambridge Road, Chicago     |
```

Straightforward, isn't it? You take the first acceptance criterion from your list, rework it to fit the Given-When-Then template, and use a scenario outline to include all the relevant examples you collected during the previous phase of your analysis.

The increased precision level allows your team to spot a possible edge case: what if a restaurant has two establishments in two different locations in the same city? There might be one Deep Lemon in Reno at 6750 South Street and a second one at 289 Laurel Drive, for example.

Should you allow that in your application? And if the answer is yes, should you make that process easier? In the end, you decide there's nothing wrong with accepting multiple locations, but you don't have time to optimize the process. Users will have to make do with what they have. To finalize the decision, you add another example to the outline.

Listing 6.4 [BETTER] Second draft of the executable specification

```
Feature: New businesses

  Scenario Outline: Businesses should provide required data

    Given a restaurant <business> on <location>
     When <business> signs up to Mapper
     Then it should be added to the platform
      And its name should appear on the map at <location>

    Examples:
      | business      | location                          |
      | Deep Lemon    | 6750 Street South, Reno           |
      | Deep Lemon    | 289 Laurel Drive, Reno            |
      | Matt's        | 9593 Riverside Drive, St. Louis   |
      | Back to Black | 8114 2nd Street, Stockton         |
      | Green Pencil  | 8583 Williams Street, Glendale    |
      | Le Chef       | 3318 Summit Avenue, Tampa         |
      | Paris         | 2105 Briarwood Court, Fresno      |
      | Christie's    | 714 Beechwood Drive, Boston       |
      | The Monument  | 77 Chapel Street, Pittsburgh      |
      | Anchor        | 110 Cambridge Road, Chicago       |
```

First location of a business

Second location of the same business

This is a fine example of a typical SBE tendency: tests guiding implementation. Only when you get to the precision level of test cases can you see that you overlooked an important element of the design. That's because tests and requirements are essentially connected. In 1968, Alan Perlis wrote that "a simulation which matches the requirements contains the control which organizes the design of the system."[2] A test predefines "ideal" outputs and inputs up front; the application code must then be designed so that the *real* inputs and outputs match the ideal ones defined by the test. Otherwise, the test will fail.

[2] *Software Engineering: Report on a conference sponsored by the NATO SCIENCE COMMITTEE, Garmisch, Germany, 7th to 11th October 1968*, eds. Peter Naur and Brian Randell (Scientific Affairs Division, NATO, 1969), http://mng.bz/jn3d.

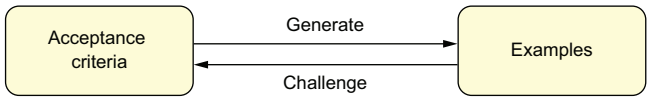


Figure 6.11 Collecting, analyzing, and refining examples is a continuous, never-ending process that exists within a powerful feedback loop.

TIP Here's a rule of thumb: good code designs usually don't need complex tests. If your tests are too complicated, you may be missing an important domain concept or tackling a known concept the wrong way.

For the second time in this chapter, the feedback mechanisms of the SBE process have led you to discover something you missed in your initial analysis. You should expect to make such discoveries multiple times during any feature's development. You'll probably go back and forth multiple times during development or even after implementation (as represented in figure 6.11). SBE practitioners should adopt the mindset that there's no such thing as a single moment when a feature is finished—features only get *released*. These are two different things.

You can begin analyzing requirements either by looking for examples, as you did in Mapper's case, or by perfecting a list of acceptance criteria. Either way, feedback lets you spot inconsistencies more quickly and easily. You shouldn't expect to get everything right the first time; that's typical.

6.5 Refining scenarios

With a scenario now in place, your team can implement it. Implementing the behaviors described by a scenario is the stage with the highest possible precision before a feature is released to customers who validate its business value in the real world. This section will show you what happens when a raw scenario first meets working code, and how that meeting increases precision to a release-ready level.

In chapters 1 and 2, you saw that after you write the first draft of a new executable specification, you often need to *refine scenarios* and *choose key examples* to improve the readability of your executable specification (see figure 6.12). That happens when teams refine their specifications to merge similar examples, reject the ones that introduce noise, and choose the most meaningful or descriptive ones.

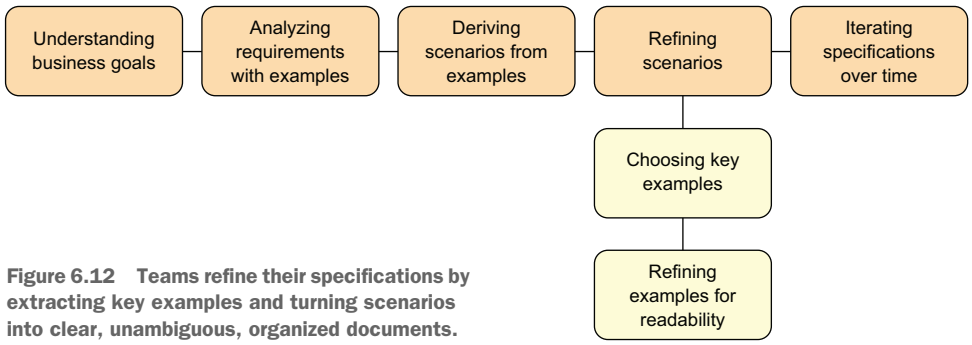


Figure 6.12 Teams refine their specifications by extracting key examples and turning scenarios into clear, unambiguous, organized documents.

Returning to Mapper, after your team begins to implement the behaviors from the scenario, they notice that the examples used in the previous section don't test any edge cases other than a business with two locations. The examples don't specify what happens when any attribute of a business is missing. All the examples in the outline end with the business successfully joining Mapper's platform. Why aren't there any counterexamples?

Let's think for a moment and look for a few counterexamples that would go astray from the happy path:

- The applicant might forget to fill out the input field with the business name on the registration form.
- The applicant might forget to mark the location on the map.
- The applicant might make both of the previous mistakes.
- The applicant might provide a location, but it might be inaccurate; for example, the user might mark the middle of a river as a location for their business.

Having defined new examples, you should now do two things. First, you need to remove redundant examples that don't bring any value to the specification. Second, add new examples and counterexamples that express the failure scenarios. You can also split the examples into multiple tables to improve readability.

Listing 6.5 [BEST] Executable specification with refined key examples

```
Feature: New businesses

  Scenario Outline: Businesses should provide required data

    Given a restaurant <business> on <location>
     When <business> signs up to Mapper
     Then it <should?> be added to the platform
      And its name <should?> appear on the map at <location>

    Examples: Business name and location should be required
      | business          | location | should?   |
      | UNNAMED BUSINESS | NOWHERE  | shouldn't |

    Examples: Allow only businesses with correct names
      | business          | location                  | should?   |
      | Back to Black     | 8114 2nd Street, Stockton | should    |
      | UNNAMED BUSINESS | 8114 2nd Street, Stockton | shouldn't |

    Examples: Allow businesses with two or more establishments
      | business    | location                | should? |
      | Deep Lemon | 6750 Street South, Reno | should  |
      | Deep Lemon | 289 Laurel Drive, Reno  | should  |

    Examples: Allow only suitable locations
      | business | location                     | should?   |
      | Anchor    | 110 Cambridge Road, Chicago | should    |
      | Anchor    | Chicago River, Chicago      | shouldn't |
      | Anchor    | NOWHERE                     | shouldn't |
```

Failure scenario with no name or location

Failure scenario with no name

Failure scenario with an inaccurate location

Failure scenario with no location

TIP You may have noticed that in listing 6.5, `UNNAMED BUSINESS` and `NOWHERE` are uppercase. To be honest, this isn't a Gherkin convention; but the capital letters make these examples stand out, which improves readability.

The reworked outline is much easier to read and has more-comprehensive scenarios. You need only to glance at it to recognize what it tests and why. It clearly distinguishes between success examples and failure examples. To my eye, the precision level of this scenario looks like it's release ready. Congratulations!

6.6 Iterating specifications over time

You don't stop working on an executable specification after you write it (see figure 6.13). It's a continuous process. The team should validate the specification suite frequently to spot any integration errors as soon as possible, keeping the suite consistent at all times. This section will show you how.

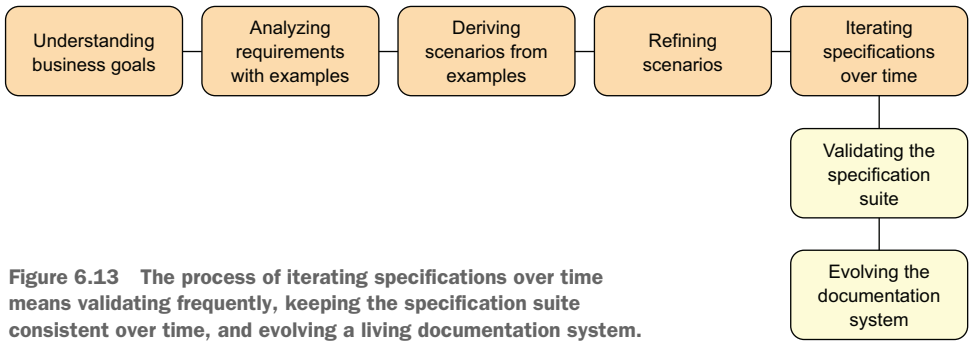


Figure 6.13 The process of iterating specifications over time means validating frequently, keeping the specification suite consistent over time, and evolving a living documentation system.

When the specifications are consistent and up to date, they evolve into a living documentation system that acts as a single source of truth about the system's behaviors. Everybody on the team can use the system freely to solve their disagreements. We'll enlarge on that, too.

6.6.1 Validating the specification suite

In the heat of the battle, while revising your scenario outline, you almost forgot that the user story you wrote at the beginning of this chapter specified two acceptance criteria:

- Every new business should provide a name and a location to display on the map.
- Every business should provide specific business hours for each day of the week.

So far, you've only taken care of the first criterion. Let's use what you've learned to write the second scenario.

Listing 6.6 Adding another scenario to the specification

```
Feature: New businesses

  Scenario Outline: Businesses should provide required data

    [...]

  Scenario Outline: Businesses should be able to set their hours

    Given a restaurant <business> on <location>
     When it schedules its hours to be <times> every day
     Then the hours should appear on the map at <location>

    Examples: Restaurants
      | business    | location                    | times      |
      | Deep Lemon | 6750 Street South, Reno     | 7 AM-8 PM |

    Examples: Bistros
      | business    | location                    | times      |
      | Le Chef     | 3318 Summit Avenue, Tampa | 9 AM-9 PM |

    Examples: Pubs
      | business    | location                    | times      |
      | Anchor      | 77 Chapel Road, Chicago     | 3 PM-3 AM |
```

After you agree on the shape of the scenario, the team proceeds to automate it. They write new application code and generate the step definitions required to test the code. Having done that, they run the test-execution engine to make sure the modified system works as they expect it to.

And that's when they find out that implementing the behavior from the new scenario breaks another scenario in the specification suite.

Listing 6.7 Broken scenario

```
Feature: Show sightseeing objects on the map

  Scenario: Tourists should be able to see sightseeing objects

    Given a sightseeing object:
      | name               | location   |
      | Memorial Monument | Oak Street |
     When Janet, who is a tourist, looks at Oak Street
     Then she should see Memorial Monument on the map
```

It turns out that the new attributes of business hours don't work well with other types of entities that Mapper features on its maps, such as sightseeing objects. Not all sightseeing objects—such as monuments—have opening and closing times.

Your team forgot about that, and the validations they added prevented sightseeing objects from being created in the database. To fix that, you decide to make the validations optional instead of required. As soon as you do, the system starts working again, and the feature is ready to be deployed to production.

Before we move on, let's dissect what happened. The specification suite has to be consistent. When the team implements the behaviors from a new executable specification,

they should execute the existing specifications in order to check whether they still work. The team must test the existing specifications every time changes are introduced to the system.

If a scenarios breaks after you introduce a new feature, you can take only two actions:

- Update the broken scenario so it complies with the changes.
- Change the new feature so it won't break the scenario.

As you can see, whereas new scenarios influence old scenarios, the old ones can also affect new features (see figure 6.14). It's another feedback loop within the process of SBE. I've already talked about it in chapter 1, which listed *validating frequently* as one of SBE's key practices.

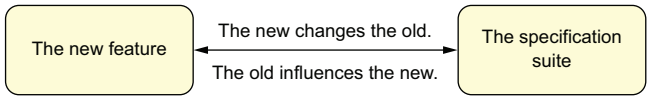


Figure 6.14 The feedback loop between new features and the existing specification suite

When you validate frequently, you once again operate in the product-critique quadrant of the testing matrix. As soon as you automate the critiques, the test-execution engine will check the application regularly against new examples, protecting the quality of your product (see figure 6.15). Modern development practices take advantage of that in various ways. For example, teams that employ continuous integration (CI) practices will integrate as often as possible, leading to multiple integrations per day. Each integration will then be verified by an automated build that can detect errors almost instantly. Some teams trust their specification suites so much that they let every change be automatically deployed to production if the tests pass—a practice known as *continuous deployment*.

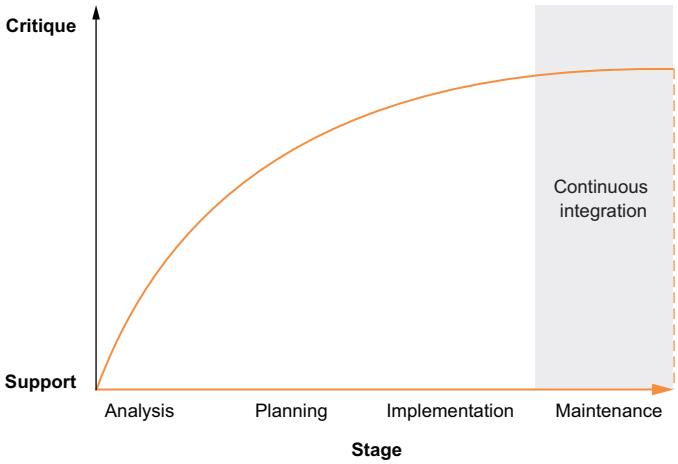


Figure 6.15 After release, examples become automated tests that critique the finished product, if they're validated frequently.

Most often, integration tests are followed by *user acceptance testing* (UAT). A popular argument for performing UAT manually is that getting your hands on the product activates a different type of perception and judgment than thinking about automation. Manipulation is different than cogitation. For example, when you test-drive a car, you notice things you wouldn't spot when poring over its specs, like the seats being too stiff or the leather not looking right.

> **DEFINITION** *User acceptance testing (UAT)*—The last phase of the software-testing process. During UAT, actual end users test the product to make sure it can handle the tasks required by the specifications.

But when business stakeholders trust in their executable specifications, they can replace simple, manual, boring checks with automated tests from the specification suite, streamlining the UAT process. (I'm not saying they should remove manual tests altogether; they can just have fewer trivial ones.) Such trust is an ultimate sign that you're doing SBE well and that the stakeholders understand why examples and scenarios are important.

6.6.2 Evolving the documentation system

After you deploy the feature to production, it starts living a life of its own. A bug may occur from time to time; you fix it, write a regression test, and move on. As with any other feature, when development ends, maintenance begins. This section covers what happens with an executable specification at the end of its life cycle, when it reaches the highest precision level.

The Mapper features turn out to be a success. Gastronomy businesses sign up like crazy. You hope that implementing the user stories about other types of businesses will happen in the future. For now, management is happy. You're proud of your team, too. The code is good; the specification looks fine.

At least, you think so, until your team comes back to the specification two months later. That's when Martha, a fresh hire on your team, takes on a new user story connected to small businesses. To implement it, she needs to understand the feature better, so she reads the specification the team created a few months ago. She still has some questions, though, so she talks to you:

"Hey," she says. "What are popular hours?"

"No idea. Why?"

"Here's a user story you created two months ago, before I joined the team. It only mentions that you might also want to consider letting gastronomy businesses specify popular hours."

"OK … that does ring a bell. But I'm not sure …"

"If it helps, I read the specification, and it already has a scenario that allows businesses to schedule some kind of hours. Maybe somebody already implemented that user story but forgot to mark it as done."

"Let's ask Gus. I think he was the last one to work with this feature. Hey, Gus, have you already implemented something called popular hours?"

"Didn't we do that two months ago? Or wait, maybe it was business hours, not popular hours. Let me check the code …"

You get the gist.

Two months ago, you made a frequent development mistake: mid battle, you thought the specifications you wrote were perfectly clear, because you still had all the domain concepts in your short-term memory. After the dust settled, you realized that your feeling of clarity was illusionary. Martha bravely brought a fresh perspective that helped you realize that issue. You cringe at the thought of how many other decisions were made without clear distinctions between domain concepts like the one she noticed.

You decide to rewrite the scenario in question and include some clarifying definitions.

Listing 6.8 [BETTER] Executable specification with clarifications

```
Feature: New businesses

  Scenario Outline: Businesses should provide required data
    [...]

  Scenario Outline: Businesses should be able to set relevant hours

    BUSINESS HOURS define when a business opens and closes.

    Businesses provide POPULAR HOURS to help their customers
    decide when it's the best time to come in.

    Given a restaurant <business> on <location>
     When it schedules <hours> to be <times>
     Then the <hours> should appear on the map at <location>

    Examples: Restaurants
      | business   | location                 | hours          | times      |
      | Deep Lemon | 6750 Street South, Reno  | business hours | 7 AM-8 PM  |
      | Deep Lemon | 6750 Street South, Reno  | popular hours  | 3 PM-5 PM  |

    Examples: Bistros
      | business   | location                 | hours          | times      |
      | Le Chef    | 3318 Summit Avenue, Tampa| business hours | 9 AM-9 PM  |
      | Le Chef    | 3318 Summit Avenue, Tampa| popular hours  | 8 PM-9 PM  |

    Examples: Pubs
      | business   | location                 | hours          | times      |
      | Anchor     | 77 Chapel Road, Chicago  | business hours | 3 PM-3 AM  |
      | Anchor     | 77 Chapel Road, Chicago  | popular hours  | 9 PM-2 AM  |
```

Added definition for business hours

Separate definition for popular hours that explains the difference

Examples of business hours

Examples of popular hours

Building and evolving a documentation system is the last step in the life cycle of any executable specification. Your work on a specification is rarely finished after you deploy

the new feature to production. You'll likely come back to rewrite the steps, or change the structure of the scenarios, or, as in the case we just discussed, add clarifications to the specification layer. Chapter 7 goes into depth on these topics.

Gall's Law

Why do I keep talking about gradual evolution of requirements and domain concepts instead of trying to find the perfect system design from scratch? Gall's Law is the reason.

Gall's Law is a rule of thumb for systems design that comes from John Gall's book *Systemantics: How Systems Really Work and How They Fail* (General Systemantics Press, 2002):

> *A complex system that works is invariably found to have evolved from a simple system that worked. A complex system designed from scratch never works and can't be patched up to make it work. You have to start over with a working simple system.*
>
> —Gall's law

I'm a strong believer in the power of this law. That's why I keep repeating that you should look for simple things that work, in terms of both the requirements and the implementation, and then build on them. Moreover, SBE's feedback loops will help you spot systems that work, and thus never break, and systems that don't work, and thus break constantly.

As the application changes and you discover new requirements, you may realize that some of the scenarios you thought were distinct are parts of a bigger whole and should be combined in a single specification. Sometimes the scenarios you thought were connected will branch out into their own requirements. As the business evolves, your specification suite should evolve with it. Changes in a specification suite often directly reflect the changes in a delivery team's understanding of the business domain. It's a fascinating subject that we'll explore deeply in chapters 8–11.

Before we finish this chapter, figure 6.16 takes another look at the full life cycle of any executable specification. The next chapter focuses on the last box in this life cycle. As you may remember from chapter 1, fully fledged executable specifications are also called *living documentation*. Living documentation is always up to date because it changes alongside the system, thanks to the link between the documentation and automated tests. When an executable specification evolves into living documentation, its *active* life cycle ends. That doesn't mean the specification won't change anymore, of course, but it'll be more *passive* from now on. Usually, it'll be changed and influenced by new specifications that enter the specification suite as the product matures. In chapter 7, which talks about the details of building a living documentation system, I discuss techniques that help you manage and maintain specifications in the passive stage of their life cycle (see figure 6.16).

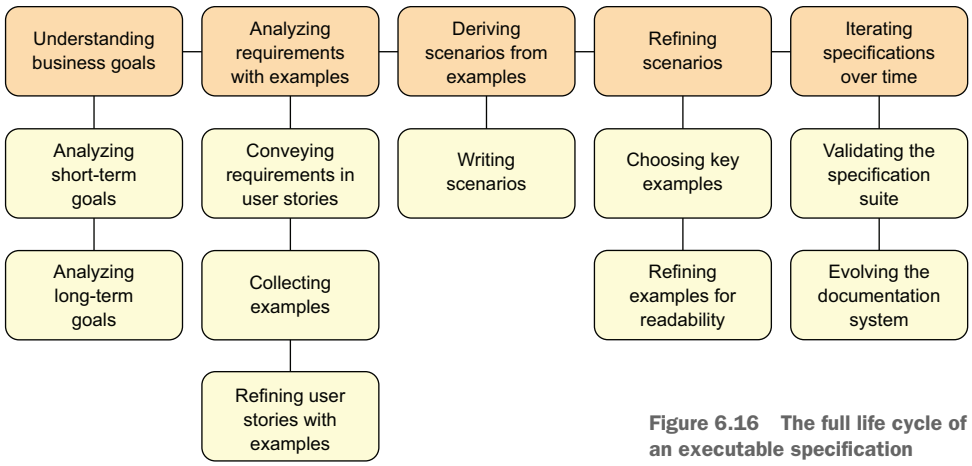


Figure 6.16 The full life cycle of an executable specification

6.7 Summary

- An executable specification evolves throughout a project's life cycle.
- As the project progresses, executable specifications become more precise. The later the life cycle's phase, the greater the need for detail.
- Exploring examples is a process of discovery: you start with little certainty about the examples' completeness, and as you contest them and improve the list, you become more certain that your understanding is sound.
- Key examples should be chosen to illustrate the acceptance criteria clearly and completely. As acceptance criteria change, the list of key examples evolves.
- Some examples support the team in their attempts to write new code, and some examples critique the product, aiming to improve its quality.
- New features add new specification documents to the specification suite, but the existing specification suite can also cause changes in new features. The new influences the old, but the old can also change the new.

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]