Gojko Adzic

Specification by Example

How successful teams deliver the *right* software

MANNING

Table of Contents

8

Refining the specification

> In its rough form, a diamond is a lusterless, translucent crystal that resembles a chip of broken glass. For it to be transformed into a jewel, it must be cut into a particular gem shape and then polished, facet by facet.
>
> —Edward Jay Epstein, *The Diamond Invention*[1]

Collaborative discussion is a great way to build a shared understanding, but that isn't enough to drive any but the simplest of projects. Unless the team is very small and the project is very short, we need to record this knowledge in a way that doesn't depend on peoples' short-term memory.

Taking a photo of the whiteboard after a discussion on key examples is a simple way to capture this knowledge, but the examples are just raw material. Raw examples are like uncut diamonds—very valuable but not nearly as much as in a processed form. Separating real diamonds from rock, polishing them, and breaking them into sizes that are easy to sell increases the value significantly. The same can be said for the key examples we use to illustrate a requirement. They're a great starting point, but in order to get the most value out of them we have to refine them, polish them to show the key points clearly, and create specifications that teams can use both now and in the future.

One of the most common reasons for failing with Specification by Example is not taking the time to process these raw examples. Discussion about specifications often leads to experimentation. We discover new insights and restructure examples to look at them from a higher level of abstraction. This results in some great examples but also a lot of dead ends and rejected ideas. We don't necessarily need to capture all these intermediary examples or record how we got to the result.

[1] http://www.edwardjayepstein.com/diamond/chap11.htm

On the other hand, just recording the key examples we want to keep without any explanation won't allow us to communicate the specification effectively to anyone who hasn't participated in the discussions.

Successful teams don't use raw examples; they *refine the specification* from them. They extract the essence from the key examples and turn it into a clear and unambiguous definition of what makes the implementation complete, without any extraneous detail. This acceptance criterion is then recorded and described so that anyone can pick up the resulting specification and understand it at any time. This specification with examples captures the conditions of satisfaction, the expected output of a feature, and its acceptance test.

Specifications with examples are acceptance tests

A good specification, with examples, is effectively an acceptance test for the described functionality.

Ideally, a specification with examples should unambiguously define the required functionality from a business perspective but not how the system is supposed to implement it. This gives the development team the freedom to find the best possible solution that meets the requirements. To be effective in these goals, a specification should be

- Precise and testable
- A true specification, not a script
- About business functionality, not about software design

Once the functionality is implemented, the specification that describes it will serve a different purpose. It will document what the system does and alert us about functional regression. To be useful as long-term functional documentation, the specification has to be written so that others can pick it up months or even years after it was created and easily understand what it does, why it's there, and what it describes. To be effective in these goals, a specification should be

- Self explanatory
- Focused
- In domain language

This chapter focuses on how to refine specifications to achieve all these goals. But first, to put things into a more concrete perspective, I show examples of good and bad specifications. At the end of this chapter, we'll refine the bad specification by applying the advice given in this chapter.

Chapter 8. Refining the specification. Specification by Example: How successful teams deliver the right software, ISBN: 9781617290084

An example of a good specification

An example of a very good specification with examples is shown here.

Free delivery

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

Examples

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

This specification is self-explanatory. I often show this example to people at conferences and workshops, and I've never had to say a single word to explain it. The title and the introductory paragraph explain the structure of the examples so that readers don't need to work back from the data to understand the specified rule. Realistic examples are also there, to make the specification testable and explain the behavior in edge cases, for example, what happens when someone buys exactly 10 books.

This is a specification, not a script for how someone might test the examples. It doesn't say anything about application workflow or session constraints. It doesn't explain how the books are purchased, just what the available delivery mechanism is. It doesn't try to talk about any implementation specifics. That's all left to the developers to work out in the best way possible.

This specification is focused on a particular rule for free delivery. It includes only the attributes relevant for that rule.

An example of a bad specification

Compare the previous specification to the example shown in figure 8.1. This is a great example of a very bad specification.[2]

Simple Acceptance Test for Payroll

First we add a few employees

Employees			
id	name	address	salary
1	Jeff Languid	10 Adamant St; Laurel MD 20707	1005.00
2	Kelp Holland	128 Baker St; Cottonmouth, IL 60066	2000.00

Next we pay them.

Pay day	
pay day	check number
1/31/2001	1000

We make sure their paychecks are correct. The blank cellls will be filled in by the Paycheckinspector fixture. The cells with data in them already will be checked.

Paycheck inspector				
id	amount	number	name	date
1	1005			
2	2000			

Finally we make sure that the output contained two, and only two paychecks, and that they had the right check numbers.

Paycheck inspector
number
1000
1001

Figure 8.1 A confusing specification

Although it has a title and some text around the tables, seemingly to explain what's going on, the effect of that is marginal. Why is this document called "simple"? It's payroll related, obviously, but what exactly is it specifying?

It's not really clear what this document is specifying. We need to work backwards from the test data to understand the rules. It seems to verify that the checks are printed with unique numbers, starting from a number that's given as a parameter. It also seems to validate the data printed on each check. It also explains in words that one check is printed per employee.

[2] This example is from a real project and was previously included with FitNesse. We used it in a workshop on refining the specifications in June 2010 in London. As a result, the example was changed in the FitNesse distribution.

This document has a lot of seemingly incidental complexity—names and addresses aren't really used anywhere in the document apart from the setup. Database identifiers appear in the tables, but they're irrelevant for the business rules. The database identifiers are used in this example to match employees with the Paycheck Inspector, introducing technical software concepts into the specification.

The Paycheck Inspector was obviously invented just for testing. When I read this for the first time, I imagined Peter Sellers in a Clouseau outfit inspecting checks as they go out. I'm sure that this isn't a business concept.

Another interesting issue is the blank cells in the assertion part of this specification, and the two Paycheck Inspector tables seem unrelated. This example is from FitNesse, and blank cells in that tool print test results for troubleshooting without checking anything. That effectively makes this specification an automated test that a human has to look over—pretty much defeating the purpose of automation. Blank cells in FitNesse are typically a sign of instability in tests, and they're a signal that something is missing. Either the automated test is hooking into the system in the wrong place, or an implicit rule is hidden there that makes the test results unrepeatable and unreliable.

The language used in the specification is inconsistent, which makes it hard to make a connection between inputs and outputs. What is the 1001 value in the table at the bottom? The column header tells us that it's a number, which is a technically correct but completely useless piece of information. The second box has a check number, but what kind of a number is that? What's the relationship between these two things?

Presuming that the addresses are there because checks are printed as part of a statement with an address for automated envelope packaging, the test based on this specification fails to verify at least one very important thing: that the right people got paid the right amount. If the first person got both checks, this test would happily pass. If they both got each other's salaries, this test would pass. If a date far in the future was printed on the checks, our employees might not be able to cash them, but the test would still pass.

Now we come to the real reason for the blank cells. Ordering of checks is not specified. This is a functional gap that makes the system hard to test in a repeatable way. The author of this FitNesse page decided to work around that technical difficulty in the specification, not in the automation layer, and created a test that gives false positives.

Without more context information it's hard to tell whether this test is verifying one thing only. If the check printing system is used for anything else, I'd prefer to pull out the fact that check numbers are unique and start from a configured value in a separate page. If we only print salary checks, it's probably part of salary check printing.

We'll refine this horrible document later in this chapter. But first, let's first go over what makes a good specification.

What to focus on when refining specifications

In the introduction to this chapter I laid out some goals for good specifications. Here are some good ideas on how to achieve those goals.

Examples should be precise and testable

A specification needs to be an objective measure of success, something that will unambiguously tell us when we're finished with development. It has to include verifiable information—combinations of parameters and expected outputs that can be checked against the system.

In order to satisfy these criteria, a specification has to be based on precise realistic examples. See the "Examples should be precise" section in chapter 7 for some good techniques on how to ensure that the examples are precise.

Scripts are not specifications

Business users will often think about performing an action through the user interface or through several steps, explaining how they'd use the system to achieve something instead of what the system is supposed to do. Such examples are scripts, not specifications.

A *script* explains how something can be tested. It describes business functionality through lower-level interactions with a system. A script requires the reader to work back from the actions and understand what's really important and what exactly is being illustrated. Scripts also bake the test into workflow and session constraints, which might change in the future even when the underlying business rules don't change.

A *specification* explains what the system does. It focuses on the business functionality in the most direct way possible. Specifications are shorter because they describe the business concepts directly. That makes them easier to read and understand than scripts. Specifications are also a lot more stable than scripts, because they won't be affected by changes in workflow and session constraints.

Here's an example of a script:

1. Log on as user *Tom*.
2. Navigate to the *home page*.
3. Search for *Specification by Example*.
4. Add *first* result to shopping cart.
5. Search for *Beautiful Testing*.
6. Add *second* result to shopping cart.
7. Verify that number of items in cart is *2*.

This script tells us *how* something is done but doesn't directly explain *what* we're specifying. Take a piece of paper and try to write down what exactly this example is specifying before continuing to read the next paragraph. Were you able to write down anything at all? If so, do you think that's the only thing that the example could possibly describe?

There are so many possibilities for what this example describes. One option is that multiple items can be added to the shopping cart. An equally possible option is that the shopping cart is empty after a user logs on. A third option is that the first search result for Specification by Example and second search result for Beautiful Testing can be added to the shopping cart.

This is a very precise and testable example; we can execute it and confirm whether the system gives us the expected result or not. The problem with this script is that it doesn't contain any information on what functionality it actually represents. The people who wrote it might know exactly what it's supposed to do when they implement the functionality for the first time. Six months later, that will no longer be obvious.

This script isn't a good communication tool. We can't really tell what this is about or know which part of the system is wrong. If the test based on this script suddenly starts failing, someone will have to spend a lot of time analyzing many different areas of code.

The step in which Tom logs on at the start is most likely required because of workflow constraints of the website. Unless this example illustrates a business rule related to this user in particular, the fact that Tom is the person who logs on is irrelevant. If his user account is disabled for any reason, this test will start failing, but the system might not necessarily have a problem. Someone will have to waste a lot of time to discover that.

Capturing acceptance criteria with scripts instead of specifications costs a lot of time in the long term, and we can save this time if we take a few minutes to restructure the examples up front. For an example of how to refine such scripts to more useful specifications, see the "Refining in practice" section at the end of this chapter. Rick Mugridge and Ward Cunningham have a lot of good advice on restructuring scripts to be better specifications in *Fit for Developing Software* (Prentice Hall, 2005).

Don't create flow-like descriptions

Watch out for descriptions of flows (first do this, then do that, ...). Unless specifying a genuine process flow, this is often a sign that a business rule is illustrated using a script. Such scripts will cause a lot of long-term maintenance problems.

Watch out for descriptions of *how* the system should work. Think about *what* the system should do.

Ian Cooper's team at Beazley realized that about six months after they started implementing Specification by Example. During a team retrospective, they started arguing that their acceptance tests are too costly to maintain. Looking for ways to reduce the cost, they restructured scripts into specifications. Cooper said:

> Models we had for doing tests were the same as manual tests, translated into scripts. Our early tests were following a scripting approach; the test was a sequence of things with some checks at the end. Once we changed over to "what should it do," it became a lot easier.

Describing acceptance tests as scripts instead of specifications is one of the most common mistakes teams make early on. Scripts work relatively well as a development target with short iterations, because people still remember what the script describes when they implement it for the first time. But they're hard to maintain and understand later. It can take several months for this problem to show up, but when it does, it will hurt badly.

Specifications should be about business functionality, not software design

Ideally, a specification should not imply software design. It should explain the business functionality without prescribing how it's going to be implemented in software. This serves two purposes:

- It allows developers to find the best possible solution now.
- It allows developers to improve the design in the future.

Specifications that focus on business functionality, without describing the implementation, enable the implementation to change more easily. A specification that doesn't say anything about software design won't need to change when the design improves. Such specifications facilitate future change by acting as an invariant. We can run the tests unmodified based on those specifications after we improve the software design, to ensure that all the previous functionality is still there.

Avoid writing specifications that are tightly coupled with code

Specifications that are tightly coupled with code and closely reflect the software implementation result in tests that are brittle.

Changes in software design break such tests, even when the business functionality described by the test doesn't change. Specifications with examples that produce brittle tests

introduce additional maintenance costs instead of facilitating change. Aslak Hellesøy points this out as one of the key lessons he learned about Specification by Example:

> We wrote too many acceptance tests, and sometimes they were too tightly coupled with our code. Not quite as coupled as unit tests would have been, but still coupled. In the worst case it would take up to eight hours after a big refactoring to update the test scripts. So we learned a lot about striking a good balance between how many tests you have and how you write them.

Watch out for names and concepts in specifications that come from software implementations and do not exist in the business domain. Examples are database identifiers, technical service names, or object class names that aren't first-order domain concepts, and concepts invented purely for automation purposes. Restructure the specifications to avoid these concepts, and they'll be much easier to understand and maintain long term.

Technical tests are important, and I'm not arguing against having such tests that are closely coupled with the software design. But such tests should not be mixed with executable specifications. A common mistake for teams starting with Specification by Example is to drop all technical tests, such as the ones at the unit or integration level, and expect that executable specifications will cover all aspects of the system. Executable specifications guide us in delivering the right business functionality. Technical tests ensure that we look at low-level technical quality aspects of the system. We need both, but we shouldn't mix them. Technical test automation tools are much better suited for technical tests than the tools we use to automate executable specifications. They'll enable the team to maintain such tests much easier.

Resist the temptation to work around technical difficulties in specifications

When: Working on a legacy system

Legacy systems often have lots of technical quirks, and they're hard to change. Users have to work around these technical difficulties, and it becomes difficult to distinguish the real business process from workarounds.

Some teams fell into a trap by including these process workarounds in their specifications. This binds the specifications not only to the implementation but also to those technical issues. Such specifications are ineffective as a facilitator of change in legacy systems. They quickly become expensive to maintain. One small change in code might require hours of updating executable specifications.

Johannes Link worked on a project where about 200 different objects had to be constructed to run the basic test scenarios. Those dependencies were specified in the executable specifications, not pushed to the automation layer. A year later, test maintenance became so costly that the team was pushing back on changes. Link says:

> Changing one feature broke a lot of tests. They couldn't afford to implement some new requirements, because this would have been too costly in terms of the tests, and they knew they needed the tests to keep the bug rate low.

Most automation tools[3] for executable specifications separate the specification from the automation process (more on this in the "How does this work?" tip at the beginning of chapter 9). The specification is in human-readable form, and the automation process is captured using a separate automation layer of programming language code.

Solve technical difficulties in the automation layer. Don't try to solve them in the test specifications.

This will allow you to change and improve the system more easily. Solving technical difficulties in an automation layer allows you to benefit from programming language features and tools when describing and maintaining technical validation processes. Programmers can apply techniques and tools to reduce duplication, create maintainable code, and easily change it. If the technical workarounds are contained in the automation layer, the specifications will be unaffected when you improve the technical design and the workarounds are no longer required.

Pushing technical workflows into the automation layer also makes specifications shorter and easier to understand. The resulting specification will explain the business concepts at a higher level of abstraction, focusing on the aspects that are important for a particular set of examples (more on this in the "Specifications should be focused" section later in this chapter).

[3] See http://specificationbyexample.com for more information on tools that support automation.

Don't get trapped in user interface details

When: Web projects

When starting out with Specification by Example, many teams wasted a lot of time by describing irrelevant examples of minor user interface details. They were following the process of Specifying by Example for the sake of the process, not to extend their understanding of the specification.

The user interface is visual, so it's easy to think about. I've seen projects where teams and customers spent hours describing navigation menu links. But that part of the user interface carried virtually no risk, and that time could have been spent discussing much more important functions.

Phil Cowans had a similar experience at Songkick when they started implementing Specification by Example, and he thinks about that as one of the key early mistakes.

> Early on we spent too long testing trivial bits of user interface, because that was easy to do. We didn't spend enough time digging into edge cases and alternative paths through the application. It's quite easy to test what you can see, but ultimately you need to have a deep understanding of what the software does rather than what the user interface looks like. Thinking in terms of user stories and paths through the application really helps.

Instead of dwelling on user interface details, it's more useful to think about user journeys through the website. When specifying collaboratively, invest time in parts of the specifications in proportion to their importance to the business. Items that are important and risky should be explored in detail. Those that aren't that important might not need to be specified so precisely.

Specifications should be self-explanatory

When an executable specification test fails because of a functional regression, someone has to look at it, understand what went wrong, and find out how to fix it. This can happen years after the specification was originally written, when the people who wrote it are no longer working on the same project. That's why it's important for specifications to be self-explanatory.

Of course, ensuring that a specification is self-explanatory also helps to avoid any misunderstanding when we develop the specified functionality for the first time.

Use a descriptive title and explain the goal using a short paragraph

Just a few words at the start of a specification can make a big difference and save a lot of time later.

If a specification contains only inputs and expected outputs, anyone who reads that document will have to reconstruct the business rule from the examples.

It's crucial to choose a descriptive title for a specification. The title should summarize the intent. Think about what you'd type into Google's search box to look for a specification if it were somewhere on the web, and use that as the title. This will make it easy for readers to discover the appropriate specification when searching for an explanation of a piece of functionality.

A reader also needs to understand the structure of the specification and its context. Explain the goal of the specification and the structure of examples in a few words—no more than a short paragraph—and put it in a header. A good trick for writing the description is to write only the examples first and then try to explain them to someone else. Capture what you said while explaining the examples, and put that in the header of the specification.

Show and keep quiet

When: Someone is working on specifications alone

In order to: Check whether a specification is self-explanatory

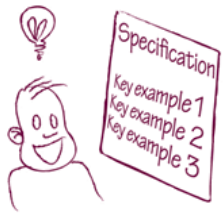
To check if a specification is self-explanatory, get someone else to look at the document and try to understand it, without you saying a word about it.

To ensure that a specification is really self-explanatory, ask the other person to explain what they understood and see if that matches your intention.

If I show a specification to someone and I find myself having to explain it, I write down the explanation and put it in the header. Explaining the examples often leads me to use more meaningful names or insert comments to make the examples easier to understand.

Don't overspecify examples

Many teams made the mistake of extending specifications to include all the possible combinations of input parameters once they put the basic automation infrastructure in place. A common explanation for this was that testers were trying to verify additional examples by reusing the automation framework.



The problem with this approach is that the original key examples get lost in a sea of other values. Specifications become hard to understand, which means that they're no longer self-explanatory.

> A specification that defines three key examples properly is much more useful than one that specifies a hundred examples poorly.

An additional problem with this approach is that executing many more examples to verify the same cases requires more time, so it slows down test execution and gives the delivery team slower feedback.

Lisa Crispin's team ran into this problem while working on automated compliance testing, with rules that are prescribed by the regulators and don't necessarily follow any logic. Crispin collaborated with her product owner to specify algorithms dealing with many permutations, so they wrote a lot of complex executable specifications several sprints ahead of development. Developers were overwhelmed when they started looking at these specifications. Crispin elaborated:

> "They [developers] looked at the tests [executable specifications] and were confused; they couldn't see the forest for the trees. They could not use the tests because they did not know what to code. So we found out that the tests should give us the big picture but not necessarily all the detail right away."

A specification should list only the key representative examples. This will help to keep the specification short and easy to understand. The key examples typically include the following:

- A representative example illustrating each important aspect of business functionality. Business users, analysts, or customers will typically define these.
- An example illustrating each important technical edge case, such as technical boundary conditions. Developers will typically suggest such examples when they're concerned about functional gaps or inconsistencies. Business users, analysts, or customers will define the correct expected behavior.

- An example illustrating each particularly troublesome area of the expected implementation, such as cases that caused bugs in the past and boundary conditions that might not be explicitly illustrated by previous examples. Testers will typically suggest these, and business users, analysts, or customers will define the correct behavior.

There is, of course, benefit in reusing the automation structure put in place for key examples to support testers who want to do more testing. Sometimes the easiest way to explore the system behavior with different boundary values is to bolt on more examples to existing specifications. This can make the specifications longer, less focused, and harder to understand.

Instead of complicating the main specification, create a separate automated test and point to it from the main one. If you use a web-based system for living documentation, you could use a web link to connect the two pages. With file-based systems, use a file path or a shortcut in the description of the specification.

The new test can use the same structure as the original specification and list many additional examples. The main specification of a feature will still be useful as a communication tool and provide quick feedback. Additional tests can explore all different combinations for the purpose of extensive testing. The primary specification can be validated at every change to provide quick feedback. Supplementary tests can run overnight and on demand to give the team confidence in all the additional cases.

Don't try to cover every single case

A common cause for overspecifying examples is a fear by analysts or customers that they will be blamed for any missing functionality. With a collaborative specification process, the responsibility for getting the specifications correct is shared, so there's no justification to do this. André Brissette, the Talia product director at Pyxis, points that out as one of the key lessons learned:

> "Decide what to cover and what not to cover depending on the conditions of success for the story. If you think that with those tests you can really cover the conditions of success, then you are OK. If that is not the case, you have a problem. If at the end of the sprint, or in the future, it turns out that you were missing something, one thing is clear: What you made was the condition of success and pretty much the contract between everybody. At that point, it [additional functionality] was not needed. As an analyst, you don't have to take the blame."

Start with basic examples; then expand through exploring

When: Describing rules with many parameter combinations

To solve the problem described in the previous section, Crispin's team decided to write only high-level specifications before starting to work on a story, leaving detailed tests for later.

A tester and a developer specified a happy path together when they started to work on a story. A developer then automated the specification and wrote the code, allowing testers to reuse the automation framework and add test cases. Testers then explored the system and experimented with different examples. If they found a failing test case, they went back to the programmers to extend the specification and get the problem fixed.

Instead of overcomplicating the specification, basic examples help drive the happy path and put the automation structure in place.

Additional examples can then be tried based on risk, and specification can be extended gradually. This is an interesting solution to handle cases where classes of equivalence aren't easy to determine at first and where the implementation drives edge cases.

Specifications should be focused

A specification should describe a single thing—a business rule, a function, or a step of a process. Such focused specifications are easier to understand than the ones that specify several related rules. A specification should also be focused on only the key attributes of examples that are important for the item it's trying to demonstrate.

Focus brings two important benefits to specifications: Focused specifications are short, so they're easier to understand than longer, less-focused ones. They're also easier to maintain. A specification that covers several rules will be influenced by changes in all of the involved areas of the system. This will cause the automated tests based on the specification to break more often. Even worse, when such a test breaks, it will be hard to pinpoint problems.

Use "Given-When-Then" language in specifications

In order to: Make the test easier to understand

As a rule of thumb, a specification should declare the context, specify a single action, and then define the expected post-conditions.

A good way to remember this is *Given-When-Then* or *Arrange-Act-Assert*. Given-When-Then is a common format for specifying system behaviors, popularized by the early behavior-driven-development articles. It requires us to write scenarios of system behaviors in three parts:

- Given a precondition
- When an action happens
- Then the following post-conditions should be satisfied

Some automation tools, such as Cucumber[4] and SpecFlow,[5] use exactly that language for executable specifications. See figure 8.1 for an example. Even with different tools that might use a tabular, keyword-based or free-form text system, structuring specifications to follow a Given-When-Then flow is an excellent idea.

Triggering a single action is crucial. This ensures that a specification is focused on only that action. If a specification lists several actions, a reader will have to analyze and understand how these actions collaborate to produce the final effect in order to understand the results.

If a set of actions is important from a business flow perspective, it's probably important enough to be given a name and used as a higher-level concept, so it should be captured in a higher-level method in the domain code. This higher-level method can then be listed in the specification.

A specification can still define several preconditions and postconditions (multiple items in the Given and Then sections) as long as they're all directly related to the function specified by the test. The following example of a Cucumber test has two preconditions and two postconditions:

Scenario: New user, suspicious transaction

Given a user with no previous transaction history,

And the user's account registration country is the UK,

When the user places an order with delivery country U.S.,

Then the transaction is marked as suspicious,

But the user sees order status as "Pending."

A potential pitfall with Given-When-Then language is that it's like prose, which often encourages people to think about flows of interactions rather than expressing business functionality directly. Use the advice from section "Scripts are not specifications" earlier in this chapter to avoid such problems.

[4] http://www.cukes.info
[5] http://specflow.org

Don't explicitly set up all the dependencies in the specification

When: Dealing with complex dependencies/referential integrity

In data-driven projects that require complex configuration, objects can rarely be created in isolation. For example, the domain validation rules for a payment method might require that it belongs to a customer, that a customer must have a valid account, and so on.

Many teams made the mistake of putting all the configuration and setup for all prerequisites into the specification. Although this makes the specification explicit and complete from a conceptual perspective, it can also make it difficult to read and understand.

In addition, any change to any of the objects or attributes in the configuration will break the test based on this specification, even if it isn't directly related to the specified rule.

Describing all dependencies explicitly can also hide data-related issues, so this is especially dangerous on data-driven projects.

Jonas Bandi worked on a project to rewrite a legacy data management system for schools, where one of the biggest problems was understanding the existing data. The team wrote specifications that were setting up the entire context dynamically. The context in the specifications was based on the understanding of the team, not on realistic data variations. The team detected many gaps and inconsistencies in requirements only when they connected the code to the data coming from the legacy system, in the middle of the iterations (see the "Examples should be realistic" section in chapter 7).

The Bekk Consulting team working on the Norwegian Dairy Herd Recording System had a similar issue but from a different perspective. Their project is also data driven, with many objects requiring complex setup. At first, they were defining the entire context in each executable specification. This required people to perfectly guess all the dependencies. If some data was missing, the tests based on the specifications would fail because of data integrity constraints, even though the code was implemented properly.

These issues can be solved better in the automation layer, not in the specification. Move all the dependencies that aren't related to the goal of the specification to the automation layer, and keep the specification focused on only the important attributes and objects. Also see the "Test data management" section in chapter 9 for some good solutions to technical data management problems.

Apply defaults in the automation layer

Push the responsibility for creating a valid object to the automation layer.

The automation layer can prepopulate objects with sensible defaults and set up dependencies so that we don't have to specify them explicitly. This allows us to focus on only the important attributes when we write specifications, making them much easier to understand and maintain.

For example, instead of requiring all address details to set up a customer and all credit card attributes to register a valid credit card, we can just specify that a user have $100 available on the card before making a payment. Everything else can be constructed dynamically by the automation layer.

Such defaults can be set in the automation layer or provided in a global configuration file, depending on whether the business users should be able to change them or not.

Don't always rely on defaults

When: Working with objects with many attributes

Although relying on sensible defaults makes specifications easier to write and understand, some teams took that approach too far. Removing duplication is generally a good practice in programming language code but not always in specifications.

If a key attribute of an example matches the default value provided by the automation layer, it's still wise to specify it explicitly, although it can be omitted.

This ensures that the specification has a full context for the readers and also allows us to change the defaults in the automation layer. Ian Cooper warns:

> Even if it [an attribute of an example] is actually the same as the default, don't rely on that. Define it explicitly. This allows us to change the default later. It also makes it obvious what is important. When you read the specifications, you can see that the example is specifying these values on a product and you can ask, "Why is this important?"

Specifications should be in domain language

Functional specifications are important to users, business analysts, testers, developers, and anyone else trying to understand the system. In order for a specification to be accessible and readable to all these groups, it has to be written in a language that everyone understands. The language used in the documentation also has to be consistent. This will minimize the need for translation and the possibility of misunderstanding.

The Ubiquitous Language (see sidebar) fits both requirements very nicely. Ensure that you use the Ubiquitous Language in specifications, and watch out for class names or concepts that seem to have been invented for the purpose of testing and sound like software implementation concepts.

Ubiquitous Language

Software delivery teams often develop their own jargon for a project, based on technical implementation concepts. This jargon is different from the jargon of business users, leading to a constant need for translation when the two groups communicate. Business analysts then act as translators and become a bottleneck for information. Translation between the two jargons often leads to a loss of information and causes misunderstanding.

Instead of letting different jargons emerge, Eric Evans suggested developing a common language as a basis for a shared understanding of the domain in *Domain Driven Design*.[†] He called this language the *Ubiquitous Language*.

[†] Eric Evans, *Domain-Driven Design: Tackling Complexity in the Heart of Software* (Addison-Wesley Professional, 2003).

By ensuring that specifications fulfill the goals laid out in this section, we get a good target for development, and we get documents that have long-term value as communication tools. They will support us in evolving the system and incrementally building up a living documentation.

Refining in practice

Let's now clean up that bad specification we saw early in this chapter and improve it. First, we should give it a nice descriptive title, such as "Payroll Check Printing," to make sure that we can find it easily later. We should also add a paragraph that explains the goal of this specification. We've identified the following rules:

- The system prints one check per employee, with the employee's name, address, and salary on the check.
- The system prints the payment date on the checks.

- Check numbers are unique, starting from the next available check number, in ascending order.
- Checks are printed for employees in alphabetic order by employee name.

A check has a payee name, an amount, and a payment date. It doesn't have a name or a salary; those are attributes of an employee. If we print checks as part of letters that will be sent out automatically, we can say that the check also has an address that will be used for automatic envelope packaging. Let's enforce the Ubiquitous Language and use these names consistently.

A combination of a name and address should be enough for us to match employees with their check—we don't need the database identifiers.

We can make the system more testable by agreeing on an ordering rule, whatever it is. For example, we can agree to print the checks in alphabetic order by employee name. We could suggest this to a customer as a way to make the specifications stronger.

To make the specification self-explanatory, let's pull out the context and put it in the header. Payroll date and the next available check number are part of the context, along with employee salary data. We should also make it explicit what the number is for, so that people who read this specification don't have to figure this out for themselves in the future. Let's call it the "Next available check number." We can also make the specification easier to understand by making the context stick out visually, to show that it prepares the data and does not verify it.

The action that gets kicked off doesn't necessarily need to be listed in the specification. A payroll run can be executed implicitly by the table that checks payroll results. This is an example of focusing on *what* is being tested instead of *how* it's being checked. There's no need to have a separate step that says, "Next, we pay them."

Paycheck Inspector is an invented concept, and it violates the Ubiquitous Language rule. This isn't a special concept in the business domain, so let's explain what it does in a way that means something. Because we want to ensure that whoever automates the validation inspects all printed checks, let's use "All checks printed." Otherwise, someone might use subset matching, and the system might print every check twice and we won't notice.

The cleaned-up version is shown in figure 8.2.

Payroll Check Printing

The system will automaticall print payroll checks:

- one check per employee, with employee's name, address, and salary on the check
- using the payroll date
- the check numbers will be unique
- starting from the next available check number in ascending order
- in the alphabetic order based on employee name

▼ PAYROLL CONTEXT

Payroll date	10/10/2010

Next available check number	1000

Employees in the system		
name	address	salary
Jeff Languid	10 Adamant St; Laurel MD 20707	1005.00
Kelp Holland	128 Baker St; Cottonmouth, IL 60066	2000.00

All checks printed in the payroll run				
check number	check date	payee	address	amount
1000	10/10/2010	Jeff Languid	10 Adamant St; Laurel MD 20707	1005.00
1001	10/10/201/	Kelp Holland	128 Baker St; Cottonmouth, IL 60066	2000.00

Figure 8.2 A refined version of the bad specification shown in figure 8.1. Note that it's shorter and self-explanatory and has a clear title.

This version is shorter and has no incidental clutter compared to the original. It's much easier to understand. After refining the specification, we can attempt to answer the question, "Are we missing anything?" We can see if the specification is complete by experimenting with input arguments and trying to think of edge cases that might represent valid inputs but violate the rules. (There's no need to consider invalid employee data, because that should be checked in another part of the system.)

One of the heuristics for experimenting with data is to use numerical boundary conditions. For example, what happens if an employee has a salary of 0? This is a valid case; an employee might have been on unpaid leave or suspended or no longer working for us. Do we still print the check? If we keep the rule "One check per employee," any employees that were fired years ago and no longer receive salaries would still get checks printed, with zeroes on them. We could then have a discussion with the business on making this rule stronger and ensuring that checks don't go out when they don't need to.

Depending on whether payroll is the only use case for check printing, we might want to refine this further and split it into several specifications. One would describe generic check printing functionality such as unique sequential check numbers. Another would describe payroll-specific functionality, such as the number of checks printed, correct salary, and so on.

It's not about the tool

Many people complain about FitNesse because of the kind of broken executable specifications shown in figure 8.1. Tools such as Concordion are intentionally built to prevent this kind of problem. Other tools such as Cucumber promote a textual Given-When-Then structure to avoid the trap of tables that are hard to understand.

Before jumping to conclusions that a certain tool is the solution for this, you should know that I've seen similarly bad specifications written with almost all the major tools. The problem isn't in the tools; likewise, the solution isn't in the tools either. The problem is mostly in teams not putting in the effort to make the specifications easy to understand. It doesn't take much more effort to refine the specification, but the result will bring a lot more value.

The benefits of refining are sometimes not obvious instantly because collaboration helps us build a shared understanding of expected functionality. That's why many teams didn't consider refining important and ended up with huge sets of documents that are difficult to understand. Refining from key examples is a crucial step, which ensures that our specifications have long-term value as communication tools and that they will create a good foundation for a living documentation system.

Remember

- Don't just use the first set of examples directly; refine the specification from them.
- To get the most out of the examples, the resulting specification should be precise and testable, self-explanatory, focused, in domain language, and about business functionality.
- Avoid scripts and talking about software design in specifications.
- Don't try to cover every single case. Specifications aren't replacements for combinatorial regression testing.
- Start with one example for each important set of cases and add examples that illustrate particular areas of concern to programmers and testers.
- Define and use the Ubiquitous Language in specifications, software design, and tests.