Gojko Adzic

Specification by Example

How successful teams deliver the *right* software

MANNING

Table of Contents

11

Evolving a documentation system

In chapter 3, I introduced the concept of living documentation and explained why it's important, without discussing how to build it. In this chapter, I cover the practices that teams used to implement a living documentation system.

A living documentation is more than a directory full of executable specification files. To experience the benefits of living documentation, we have to organize specifications so they make sense together and add relevant contextual information that will allow us to understand individual parts.

Ideally, a living documentation system should help us understand what our system does, which means that the information must be

- Easy to understand
- Consistent
- Organized for easy access

In this chapter, I present the techniques that the teams from my research used to fulfill those three goals.

Living documentation should be easy to understand

By rigorously refining the specification, as described in chapter 8, we create executable specifications that are focused and self-explanatory and make use of the domain language of a project. As a living documentation system grows, we add information to its specifications and merge or split them. Here are some useful ideas for keeping the living documentation easy to understand as it grows.

Don't create long specifications

Documentation grows as you add functionality to the underlying system; you create new specifications and extend the existing ones. Watch out for specifications that become too long as a result.

Specifications that are too long are often a sign that something is wrong. The longer a specification is, the harder it will be to understand.

Here are some examples of things that might be wrong when a specification is too long:

- The concepts aren't explained at the appropriate level of abstraction. Ask yourself, "What are we missing here?" and try to identify the missing concepts that would allow you to break the test apart. Identifying missing concepts can lead to design breakthroughs. See "Look for implied concepts" in chapter 7 for more on this.
- Instead of being focused on a single function, the specification describes several similar functions. Break it apart into separate specifications. See "Specifications should be focused" in chapter 8 for more details.
- You're describing the functionality using a script, not a specification. Restructure the information and focus it on what the system is supposed to do instead of how it's done. See "Scripts are not specifications" in chapter 8 for further information.
- The specification contains a lot of unnecessary contextual information. Clean it up by focusing on important attributes that illustrate the goal of this particular test.

Don't use many small specifications to describe a single feature

As a system evolves, our understanding of the domain changes. Concepts that start out differently might start to look similar—we discover that they are two sides of the same coin. Similarly, we might break complex concepts into smaller elements that suddenly start looking similar to existing concepts. In these cases, multiple specifications in a living documentation system that describe the same feature should be merged.

Rakesh Patel's team at Sky Network Services went too far in breaking down the specifications at one point. A single specification no longer described an entire feature. Patel said:

> If you have lots of examples in one file, it makes that file a bit more difficult to work with because you see a lot of similar code and you might be in a wrong part of it. I used to prefer having a lot of different files with different examples, but then you have a lot of files so that becomes hard to track as well.

If someone has to read 10 different specifications to understand how a feature works, it's time to think about reorganizing the documentation.

Look for higher-level concepts

In the course of adding functionality to the system, we sometimes end up with similar specifications that have only minor differences.

Step back and look at what your specifications describe from a higher level of abstraction.

Once we've identified a higher-level concept, a whole set of specifications can typically be replaced with a single specification that focuses only on the attributes that are different. This makes the information easier to understand, find, and access. Identifying missing concepts might also lead to breakthroughs in system design, similar to the process described in the "Look for implied concepts" in chapter 7.

Avoid using technical automation concepts in tests

When: Stakeholders aren't technical

Instead of creating a communication tool, some teams focused on functional regression testing with their executable specifications and wrote technical acceptance tests. This allows developers to write tests quicker, but it also makes the tests harder to read and often impossible to understand for anyone who isn't a developer. Johannes Link had such an experience on his first project using FIT:[1]

> We ended up with lots of tests with lots of duplication. Developers could understand the tests, but they were cryptic for anyone from the business side. They took longer to run and longer to maintain than just JUnit tests. We threw away some of them and rewrote them in JUnit.

[1] The first automation tool for executable specifications

Specifications described in a technical language are ineffective as a communication tool. If business users care about an executable specification, then it should be described in a language they understand. If the business users don't care about a specification, it should be captured with a technical test tool.

A living documentation system that contains technical automation concepts, such as the command to wait until a given time for a process to complete, is a signal for a team to revisit the design of the underlying software system. The need to use technical concepts in living documentation often points to problems with system design, such as reliability of asynchronous processes (see "Listen to your living documentation" at the end of this chapter).

Using technical language is acceptable only when the stakeholders are technical as well and can understand what's going on in the technical language (such as SQL queries, DOM identifiers, and so on). Note that even if the stakeholders are technical, using such technical language tends to describe how something is tested rather than what the functionality is. Although such tests might initially be quicker to write, they might cause maintenance problems in the long term. See "Scripts aren't specifications" in chapter 8 for more information.

Mike Vogel used DbFit, a database test script extension for FitNesse that I wrote, to describe acceptance tests in a project with technical stakeholders who could understand scripts. In hindsight, he thinks this was a mistake:

> In the beginning they were happy because they could quickly use DbFit and not write custom fixtures, so they had automated tests from day one. Later on, as the complexity of the solution increased, there was no time in the release plan to go back and create test fixtures to make the tests simpler and more understandable and to make the system more testable. We ended up with too brittle, too complicated tests.

Living documentation should be consistent

Living documentation is probably the longest living artifact of a project. Technologies will come and go, code will be replaced with other code, but the living documentation system describes how the business works. We'll add content to it over several months or years and we need to be able to understand it later. One of the biggest challenges for many teams was keeping the structure and the language of their living documentation consistent. Stuart Taylor explains it nicely:

> There's a danger when you start writing very clear BDD tests [executable specifications] that you'll end up with 57 ways to navigate to a page because it's different every time. It's important to keep refactoring the language and get it to a point that it's not so abstract that it's cumbersome, but it's not so detailed that it isn't BDD.

A consistent language also allows us to automate executable specifications more efficiently. For Gaspar Nagy, this is one of the key guidelines for development:

> It's very important to use consistent wording and consistent expression language for acceptance criteria. It's easier for developers to spot that it's the same structure as before and easier for automation.

To keep a living documentation consistent, we have to constantly refine it and keep it in sync with the current model that's applied to the software system. As concepts evolve in software, that needs to be reflected in the living documentation system as well. This maintenance has costs, but without it there's no living documentation.

Evolve a language

Almost all the teams ended up evolving a kind of a specification language, a set of reusable patterns for specifications. For some teams, this language evolved over several months and was often instigated by maintenance problems.

Andrew Jackman explained that the Sierra team at BNP Paribas started evolving a language when they noticed that their automation layer grew too much:

> We have so much fixture code now that it's becoming a maintenance issue. We had a lot of very specific fixtures that used a very wordy description that worked for only one test, but we tried to boil it down to a generic language. We developed a mini domain-specific language in FIT for our web tests. That has reduced a lot of fixture code.

Some teams evolved the basics of this language quickly. Rob Park's team at a large U.S. insurance provider is a good example:

> The language evolved very quickly. The first three or four fixture [automation] classes were individual. We were making it work and focusing on the conversational part. We immediately noticed some kind of duplication in the step files [automation classes] and we started moving away from that. For each story we'd have one Gherkin file [executable specification], but we had five or six story cards for the same feature.

> For the most part, the steps were very similar, so we found that having a single step file for the all of the stories that belonged to that one piece of business functionality was really better. Otherwise, even though they were one-liners, we had a lot of duplication.

Evolving a language helps reduce the cost of maintaining the automation layer because reusing existing phrases to describe new specifications leads to consistency of specifications.

The fact that a living documentation system is automated and connected directly to software ensures that the software model aligns with the business model. Because of that, evolving a language for the living documentation system is a great way to create and maintain the ubiquitous language (as discussed in chapter 8).

Base the specification language on personas

When: Web projects

Some teams described user stories through personas, especially when developing websites. In those cases, the specification language can come from the activities that different personas can perform.

Personas can help simplify executable specifications and make them easier to maintain.

The team at Iowa Student Loan aligned the language used in their specifications with personas. Tim Andersen said:

> Instead of users as an amorphous blob, we talked about different people and what their motivations to use the system are, how they use the system, and what they want to get out of it. We put names on different people. Boris was a borrower. Carrie was a co-signer.
>
> Personas helped us because they made us think about how the system needs to behave from the perspective of a user. There are a bunch of positive side effects of using personas that we didn't anticipate—for example, personas were test helpers [automation components] that were able to interact with our system at a more appropriate entry point.

Personas don't make as much sense for projects that have little user interaction. Based on the success of using personas on a previous project, Andersen tried to apply the same concept to a technical data processing system. He eventually gave up and changed the language to a process flow model.

> Data comes from multiple sources and is loaded into a phone system so people can make phone calls. The phone data is updated, and we send it back to the entities who sent it to us. It is a batch process. Nobody actually uses it; it just runs. Personas weren't a good fit. We tried to get tests defined with personas and we got blank stares from the businesspeople. So I deleted all my persona code, and we changed it to be process based using the Given-When-Then keywords. That made it a lot clearer, and it made more sense to everyone.

Evolving the ubiquitous language around the activities of user personas allows us to ensure that our understanding of what individual personas need is aligned with how they use the system. This drives the structure and language used in specifications and helps us make the documentation system consistent.

Collaborate on defining the language

When: Choosing not to run specification workshops

If you decide not to run big workshops and instead use one of the alternative approaches, make sure to collaborate on defining the language.

Christian Hassa says that collaborating on a language was one of the biggest challenges for his team:

> Building a domain language that was consistent and bound well was completely impossible without guidance. Testers wrote things that developers had to rephrase. Sometimes this was because the way testers were writing them down was unclear or not easy to bind [automate]. When the tester had written already a lot of things, we had to rephrase a lot of things. If we tried to bind [automate] the first example immediately, we would notice that it was not easy to do.
>
> It's like doing pair programming compared to doing code reviews afterwards. If you do pair programming, the pair will tell you immediately if he thinks you are doing something wrong. If you do reviews, you say: Yes, next time I'll do it differently, but this time let's leave it like this.

Instead of catching consistency problems and having to go back and fix them, Christian Hassa suggests getting developers and testers to write specifications in pairs as a way to prevent such problems. This is similar to the way a pilot and a co-pilot in an airplane work to prevent problems. The risk of someone writing bad specifications is significantly reduced because the other person validates the specifications as they're writing them and watches out for problems.

The Sierra team at BNP Paribas has a relatively stable language, one that evolved over many years, and their business analysts use this language to write new specifications themselves. To avoid inconsistent language or specifications being hard to automate, they ask the developers to review anything that has a structure significantly different from the existing specifications. When working on the Norwegian Dairy Herd Recording System, the Bekk Consulting team used a similar process. Their business users write specifications with examples, but the developers review the work and advise on how to make them more consistent with the rest of the living documentation system.

Document your building blocks

It's good practice to document the building blocks for specifications; this helps people reuse components and keep the language consistent.

Some teams have built a separate documentation area for their building blocks. At Iowa Student Loan, they have a page with all the personas. It doesn't have any assertions, but instead shows which specification building blocks are already available. The page is built from the underlying automation code, creating a living dictionary of the living documentation.

But there's an even easier way to build good documentation about your project language. When asked what advice they would give a new team member on writing a good specification, almost all the research participants suggested looking at examples of existing specifications. A nice way to document specification building blocks is to extract good representative examples from the existing set of specifications. Because these specifications are already executable, this documentation of building blocks is guaranteed to be accurate and consistent.

Because living documentation supports a team as it builds a project over a long period of time, there's a danger that parts will stay in jargons that are no longer used. One part might use a language that the team used three years ago; another might be using terminology from two years ago, and so on, depending when the specifications were originally written. This pretty much defeats the

point of having a documentation system, because we'll need to have people translate the old language into the new one.

It doesn't take a lot of effort to keep the entire documentation consistent when the language evolves, and a consistent documentation will give the team much more value over the long term.

Living documentation should be organized for easy access

Living documentation systems grow quickly. As a project moves forward, the implementation team will frequently add new specifications to it; it isn't uncommon to have hundreds of specifications in a documentation system after a few months. I interviewed several teams that had more than 50,000 checks in their living documentation systems built up over the course of several years.

For the living documentation to be useful, users have to be able to find a description of a required function easily, which means that the whole documentation set has to be nicely organized and that individual specifications have to be easy to access.

Phil Cowans says that, for him, one of the biggest lessons about living documentation was that teams should think about high-level structure early:

> We didn't think about the high-level structure of the tests. We were just adding new tests when we needed. As a result, it's hard to find which tests cover which functionality. Getting the description of what the feature set of the site is and organizing the test suite along those lines (rather than just the last thing we built) would have helped. I think that's useful in terms of developing a product and maintaining a code base that's relatively easy to understand.

If we have to spend hours trying to piece together the big picture from hundreds of seemingly unrelated files every time we want to understand how something works, we might as well read the programming language code. To get the most out of living documentation, information has to be easy to find. Here are some tips on how to do that.

Organize current work by stories

Many tools for automating executable specifications allow us to group specifications into hierarchies, either as website sections and subsections or file directories and subdirectories.

If you work with a tool for automating executable specifications, it's generally a good practice to group all them together for the work that's currently in progress.

Grouping specifications into hierarchies allows us to quickly execute all those specifications as a test pack, as suggested in "Create a current iteration pack" in chapter 10.

A user story will typically require us to change several functional areas. For example, a story about enhanced registration might affect a back office report for users and how the system does age verification. It might also require us to implement new integrations with PayPal and Gmail. All these functions should be described by separate and focused executable specifications. We also want to have a clear definition of when each story is done; everything related to a story should be grouped together to facilitate easy execution of all those tests.

See the suggested organization in figure 11.1: the Current Iteration branch.

Reorganize stories by functional areas

User stories are excellent as a planning tool, but they aren't useful as a way to organize existing system functionality. Six months after PayPal integration was implemented, the fact that it initially came into the system as part of story #128 is largely irrelevant (unless you need traceability for regulatory purposes, for example). If anyone wants to understand how PayPal integration works, they'll need to remember the exact story number so they can find it.

Most teams reorganize their executable specifications into hierarchies by functional areas once they've been implemented. This makes it easy find an explanation of a feature by navigating through the hierarchy based on business functionality.

In figure 11.1, this is shown in the branch under Feature Sets. Once story #128 is implemented, we should move the specification of how PayPal integration works into payments, change back-office user reports into user management, and so on. Organizing a living documentation system in this way enables us to quickly find all the existing examples related to MasterCard payments when we want to discuss a change request in that feature.

If you still want to know how some functionality was part of a particular story, there are tools that will allow you to cross-reference the same specification from different hierarchies.

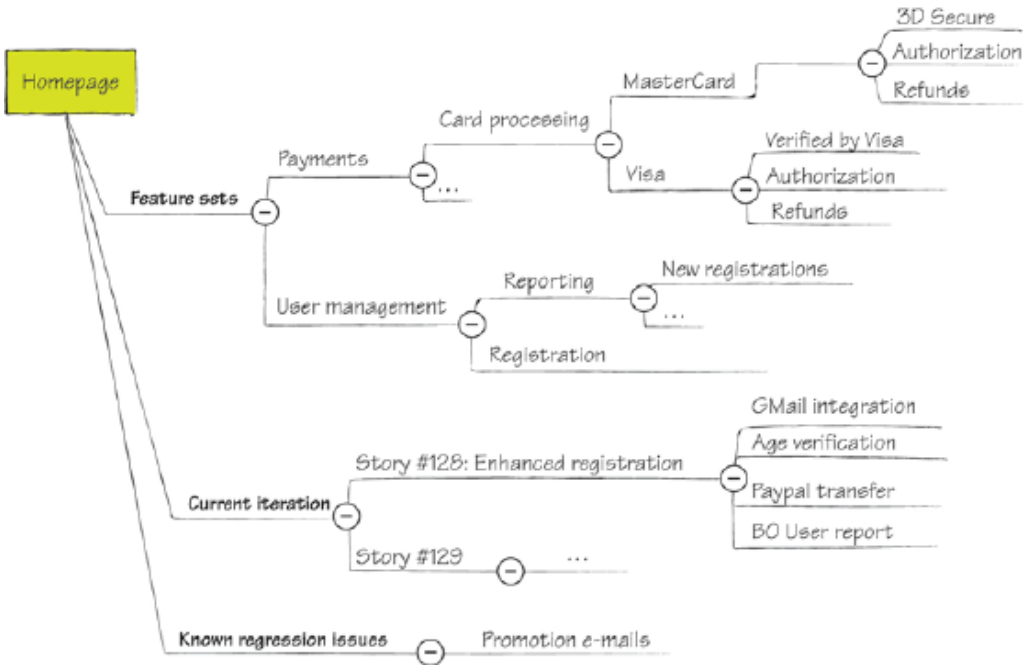

Figure 11.1 Living documentation hierarchy organized by functional areas (such as Payments and User Management). Specifications for the current iteration are organized by stories and features. Known issues waiting for more information are also in a separate holding place.

Organize along UI navigation routes

When: Documenting user interfaces

Replicate your user interface navigation structure in your living documentation system.

Ian Cooper's team at Beazley implemented an innovative organization for their living documentation system. Instead of functional areas, they replicated the user interface navigation structure in their living documentation system. Cooper says:

> FitNesse tests allowed us to pick a story and find out what's involved in a story. But it was exceptionally hard to navigate in that form. How did you know where to find the story that represented a part of the software?
>
> We restructured it so that the FitNesse page looks like a help page. I'm in this page, and I've got a FitNesse test to tell me everything I can do on this

> page. And if I click a link next to this dialog, it will take me to another page that explains the dialog. That made it much easier to find out where to go to get information on how something works.

This approach is intuitive for systems with clearly defined navigational routes, such as back-office applications. But it might cause maintenance problems if the UI navigational routes change often.

Organize along business processes

When: End-to-end use case traceability required

Structuring the living documentation system along the lines of business processes makes it easy to trace the functionality provided by a system in end-to-end use cases.

Mike Vogel worked on a software system to support pharmaceutical research where the team organized their living documentation system along the lines of business processes. He explains this approach:

> We organized our [FitNesse] tests to align with our use cases. Our use cases are organized in a hierarchy, with the top-level use case naming a system goal. Each top-level use case is also the definition of the end-to-end business processes for that goal. A use case refers to lower-level use cases, which are subprocesses.
>
> The table of contents of our requirements document is identical to the table of contents to our tests. This made it easier to understand how the tests align with the business processes. It also created direct traceability from business requirements to tests, which is critical to meet regulatory requirements in our domain.

These aren't the only ways to organize a living documentation. Another good approach is to organize information along the chapters of a help system or user guide. Use these ideas as inspiration to find the best way to set up the hierarchies for your team.

Use tags instead of URLs when referring to executable specifications

When: You need traceability of specifications

Many living documentation tools now support tags—freeform textual attributes that we can assign to any page or file. Metadata like this is generally better for traceability than for keeping specifications in a hierarchy by user stories or use cases. When the domain model changes, the living documentation should ideally follow those changes. Specifications will often get moved, merged, split, or changed. This is impossible if you rely on a strict static hierarchy for traceability, but it can easily be done if story/case numbers are assigned to specifications as tags.

Tags are also useful if you want to refer to a living documentation page from another tool, for example, from an issue-tracking-system ticket or a planning-tool schedule. Using a URL based on the current location of a page would prevent us from moving it later because the link would be broken.

Assigning a tag and linking to search results for that tag makes the system much more resilient to future changes.

Even if you don't use a web-based tool and instead keep specifications in the project directory, you can still use tags with the help of a simple script. This is what the Norwegian Dairy Herd Recording System team at Bekk Consulting did. Børge Lotre explains this approach:

> "To share Cucumber tests with customers we use Confluence and link the Cucumber tests directly from Subversion into Confluence. This prevents us from restructuring the file hierarchy of the Cucumber tests without hassle, but utilizing tags has proven to help us overcome this shortcoming. Now we use tags to document which requirements are covered by which Cucumber tests."

Avoid referring to a particular specification in the living documentation system directly, because that prevents you from reorganizing the documentation later. Metadata, tags, or keywords that you can dynamically search for are much better for external links.

A living documentation system is more than just a pile of executable specifications. Information that's buried deep inside an unmanageable list of tests is useless as documentation. To experience the long-term benefits of Specification by Example, we have to ensure that the documentation is organized in a way that makes it easy for anyone to quickly find a specification of a particular function and test it.

I've presented the most common ways of organizing the specifications here, but you don't have to stop with these. Find your own way of structuring documents that makes it intuitive for your business users, testers, and developers to find what they're looking for.

Listen to your living documentation

At first, many teams didn't understand that living documentation closely reflects the domain model of the system it describes. If the design of a system is driven with executable specifications, the same ubiquitous language and domain models will be used in both the specifications and software.

Spotting incidental complexity in executable specifications is a good indicator that you can simplify the system and make it easier to use and maintain. Channing Walton refers to this approach as "listen to your tests." He worked on an order-management system at UBS where the acceptance criteria for workflows was complex. He says:

> If a test is too complicated, it's telling you something about the system. Workflow testing was very painful. There was too much going on and tests were very complicated. Developers started asking why the tests were so complicated. It turned out that workflows were overcomplicated by each department not really knowing what the others are doing. Tests helped because they put everything together, so people could see that another department is doing validations and handling errors as well. The whole thing got reduced to something much simpler.

Automating executable specifications forces developers to experience what it's like to use their own system, because they have to use the interfaces designed for clients. If executable specifications are hard to automate, this means that the client APIs aren't easy to use, which means it's time to start simplifying the APIs. This was one of the biggest lessons for Pascal Mestdach:

> The way you write your tests defines how you write and design your code. If you need to persist patient data in a part of your test to do that, you need to make a data set, fill a data set with four tables, call a huge method to persist it, and call some setup methods for that class. That makes it really hard to come to the part where it actually tests your scenario. If your setup is hard, the tests will be hard. But then, persisting a patient in real code is going to be hard.

Markus Gärtner points out that long setups signal bad API design:

> When you notice a long setup, think about the user of your API and the stuff you're creating. This will become the business of someone to deal with your complicated API. Do you really want to do this?

Living documentation maintenance problems can also provide a hint that the architecture of the system is suboptimal. Ian Cooper said that they often broke many tests in their living documentation system with small domain code changes, an example of shotgun surgery. That led him to investigate how to improve the design of the system:

> It's an indicator that your architecture is wrong. At first you struggle against it, and then you begin to realize that the problem is not FitNesse, but how you let it interact with your application.

Cooper suggested looking at living documentation as an alternative user interface to the system. If this interface is hard to write and maintain, the real user interface will also be hard to write and maintain.

If a concept is defined through complex interactions in the living documentation, that probably means that the same complex interactions exist in the programming language code. If two concepts are described similarly in the living documentation, this probably means the domain model also contains this duplication. Instead of ignoring complex specifications, we can use them as a warning sign that the domain model should be changed or that the underlying software should be cleaned up.

Remember

- To get the most out of your living documentation system, keep it consistent and make sure that the individual executable specifications are easy to understand and easy to access for everyone, including business users.
- Evolve the ubiquitous language and use it consistently.
- As the system evolves, watch out for long specifications or several small ones that explain the same thing with minor variations. Look for concepts at a higher level of abstraction that would make these things easier to explain.
- Organize the living documentation system into a hierarchy that allows you to easily find all the specifications for the current iteration and any feature that was previously implemented.

This page intentionally left blank