Gojko Adzic

Specification by Example

How successful teams deliver the *right* software

MANNING

Table of Contents

3

Living documentation

There are two popular models today for looking at the process and artifacts of Specification by Example: the acceptance-testing-centric model and the system-behavior-specification model.

The acceptance-testing-centric model (often called acceptance test-driven development, ATDD, or A-TDD) focuses on the automated tests that are part of the Specification by Example process. In this model, the key benefits are clearer targets for development and preventing functional regression.

The system-behavior-specification-centric model (often called behavior-driven development or BDD) focuses on the process of specifying scenarios of system behavior. It centers on building shared understanding between stakeholders and delivery teams through collaboration and clarification of specifications. Preventing functional regression through test automation is also considered important.

I don't consider one of these models superior to the other; different models are useful for different purposes. The acceptance-testing-centric model is more useful for initial adoption if a team has many functional quality issues. When things are running smoothly, the behavior-specification-centric model is useful for explaining the activities of short-term and mid-term software delivery.

Preventing functional regression through test automation is the key long-term benefit of Specification by Example in both models. Although regression testing is certainly important, I don't think that's where the long-term benefits come from. First, Specification by Example isn't the only way to prevent functional regression. The team at uSwitch, for example, disables many tests after implementing the related functionality for the first time (more on this in chapter 12); they still maintain a high level of quality. Second, Capers Jones, in *Estimating Software Costs*, points out that the average defect-removal efficiency of regression testing is only 23%.[1] That can't justify the long-term investment made by successful teams to implement Specification by Example.

[1] See Capers Jones, *Estimating Software Costs: Bringing Realism to Estimating* (McGraw-Hill Companies, 2007), 509. Also see http://gojko.net/2011/03/03/simulating-your-way-out-of-regression-testing

While researching this book, I had the privilege of interviewing teams that had used Specification by Example for five years or more. Their experiences, especially those in recent years, helped me see things from a different perspective—a documentation-centric one. Many of the teams I interviewed realized that the artifacts of Specification by Example are valuable as documentation over the long term. Most discovered this after years of experimenting with ways to define specifications and tests. One of my main goals as author of this book is to present living documentation as a first-class artifact of Specification by Example. This should help readers implement a living documentation system quickly and deliberately, without years of trial and error.

In this chapter, I'll cover the documentation model and its benefits. This model focuses on business-process documentation and ensures effective long-term maintenance and support of business processes. This model is particularly useful for ensuring the long-term benefits of Specification by Example. It also prevents many common test maintenance implementation problems (more on this later in the chapter).

Why we need authoritative documentation

I've lost count of the number of times someone has given me a lengthy book about a system and included a warning that it's "not entirely correct." Like cheap wine, long paper documentation ages rapidly and leaves you with a bad headache if you try to use it a year after it was created. On the other hand, maintaining a system without any documentation also causes headaches.

We need to know what a system does to be able to analyze the impacts of suggested changes, support it, and troubleshoot. Often, the only way to find out what the system does is to look at the programming language source code and reverse-engineer the business functionality. When I interviewed Christian Hassa, owner of TechTalk, for this book, he called the process of digging out functionality from the code "system archeology." He explains a situation that will no doubt be familiar to most readers:

> We had a project where we needed to replace a legacy system. None of the stakeholders knew how certain calculations/reports were created. The users just consumed the result and trusted the old system blindly. It was horrible to reverse-engineer the requirements from the old application, and of course it turned out that some things the old system did were wrong.

Even when the undocumented code is correct, reverse engineering is an impossible task for business users, testers, support engineers, and, on most projects, even the average developer. Clearly this approach doesn't work. We need something better.

Good documentation is useful for more than software development. Many companies could benefit greatly from having good documentation about their business processes, especially as more and more businesses are becoming technology-driven. Business-process documentation is as hard to write and as costly to maintain as any kind of system documentation.

The ideal solution would be a documentation system that's easy and cheap to maintain, so that it can be kept consistent with the system functionality even if the underlying programming language code is changed frequently. The problem with any kind of comprehensive documentation is, in fact, costly maintenance. From my experience, changing the parts that are outdated doesn't contribute significantly to cost. Often, cost is the result of time spent on finding what needs to be changed.

Tests can be good documentation

Automated tests suffer from the opposite problem. It's easy to find all the places that need to be updated; automated tests can be frequently executed, and any tests that fail are obviously no longer in sync with the underlying code. But unless the tests are designed so they're easy to modify, updating them after a system change can take a lot of time. One of the pitfalls of the acceptance-test-centric approach is that it neglects this effect.

Teams that focus on tests and testing often neglect to write tests that will be easy to maintain. Over time, problems lead these teams to look for ways to specify and automate tests so they will be easier to update. Once the tests become easy to maintain, the teams start seeing many other long-term benefits from Specification by Example. Adam Knight's team at RainStor, a UK-based provider of online data-retention solutions, realized that if tests reveal the underlying purpose, they can be easily maintained. He says:

> "As you develop a harness of automated tests, those can become your documentation if you set them up properly to reveal the purpose behind them. We produced HTML reports that listed tests that were run and their purpose. Investigation of any regression failures was much easier. You could resolve conflicts more easily because you could understand the purpose without going back to other documentation."

For me, the most important point is in the last sentence: they don't have to use any other kind of documentation once the tests are clear.

Lisa Crispin of ePlan Services said that one of the biggest ah-ha moments for her was when she understood how valuable the tests were as documentation:

> We get this loan payment and the amount of interest we applied isn't correct. We think there is a bug. I can look at the FitNesse test and put in the values. Maybe the requirements were wrong, but here's what the code is doing. That saves so much time.

Andrew Jackman said that the Sierra team uses test results as a knowledge base for support:

> Business analysts see the advantage of this all the time. When someone asks where some data in Sierra comes from, they often just send the link to a test result—and it is reasonable documentation. We don't have specifications in Word documents.

I mentioned that the team at the Iowa Student Loan Liquidity Corporation used their tests to estimate the impact of a business model change and guide the implementation. The team at SongKick used their tests to guide the implementation of a system change and saved an estimated 50% of the time required as a result. I heard similar stories from many other teams.

When a team uses a piece of information to guide development, support the system, or estimate the impact of business changes, it's misleading to call that information a "test." Tests aren't used to support and evolve our systems; documentation is.

Creating documentation from executable specifications

When a software system is continuously validated against a set of executable specifications, a team can be certain that the system will do what the specifications say—or, to put it differently, that the specifications will continue to describe what the system does. Those specifications live with the system, and they're always consistent. Because we find out about any differences between specifications and the underlying system functionality right away, they can be kept consistent—at a low cost. Tim Andersen of Iowa Student Loan said that he trusts only this kind of documentation:

> If I cannot have the documentation in an automated fashion, I don't trust it. It's not exercised.

Executable specifications create a body of documentation, an authoritative source of information on the functionality of a system that doesn't suffer from the "not entirely correct" problem and that's relatively cheap to maintain. If specifications with examples were pages, the living documentation system would be the book.

A living documentation replaces all the artifacts that teams need for delivering the right product; it can even support the production of external user manuals (although probably not replace them). It does that in a way that fits nicely with short iterative or flow processes. Because we can define the specifications as we grow the underlying software system, the resulting documentation will be incremental and cheap to write. We can build business-process documentation at the same time as the supporting systems and use that documentation to evolve the software and help run the business. The world doesn't have to stop for six months while someone is compiling 500 pages of material. André Brissette from Pyxis says this is one of the least-understood parts of agile development:

> Beginners think that there is no documentation in agile, which is not true. It's about choosing the types of documentation that are useful. For people who are afraid that there is no documentation, this kind of test is a good opportunity to secure themselves and see that there is still documentation in an agile process, and that's not a two-feet-high pile of paper. This is something lighter, but bound to the real code. When you ask, "does your system have this feature?" you don't have a Word document that claims that something is done; you have something executable that proves that the system really does what you want. That's real documentation.

Most automation tools used for Specification by Example already support managing specifications over websites or exporting test results in HTML or PDF form, which is a good start for creating a documentation system. I expect there will be a lot of innovation in the tools over the next several years to assist with building up documentation from specifications with examples. One interesting project is Relish,[2] which imports specifications with examples from several automation tools and formats them to create a documentation system that's easy to use. See figure 3.1.

[2] See www.relishapp.com

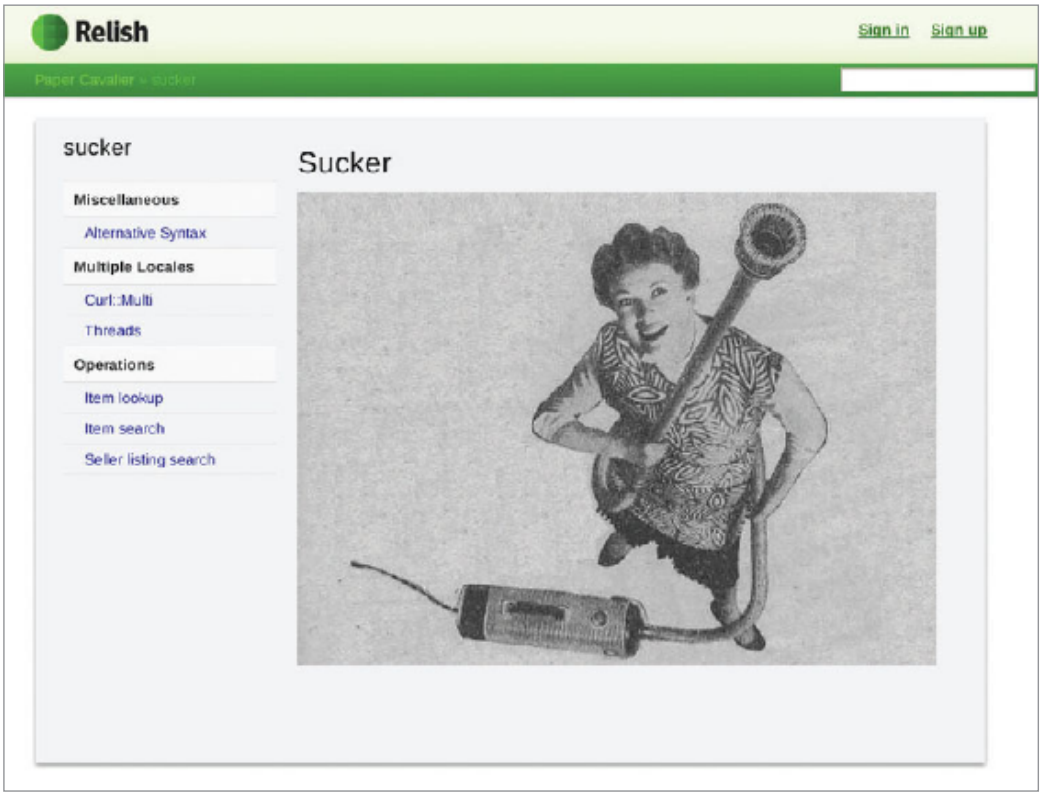

Figure 3.1 Relish builds documentation websites from executable specifications.

Benefits of the documentation-centric model

The documentation-centric model of Specification by Example should help teams avoid the most common issues with long-term maintenance of executable specifications. It should also help teams create useful documentation that will facilitate software evolution over time and help to avoid maintenance problems caused by a lack of shared knowledge.

Many teams I interviewed replaced the heart of their system or rewrote large parts of it while keeping specifications with examples and using them to guide the whole effort. This is where the investment in living documentation really pays off. Instead of spending months on system archeology and verifications, a living documentation system already provides requirements for technical updates and changes.

I think teams should consider living documentation as a separate artifact that's as important as the system they're delivering. The idea that the documentation is a key deliverable is at the core of the documentation-centric model. I expect this model resolves most of the common problems that, over time, cause teams to fail with Specification by Example. Although it hasn't been proven by any of the case studies in this book, I

consider this premise important for the future. I hope that the readers of this book will be able to get great results easier and faster because of looking at the process from this different perspective.

For example, understanding that living documentation is an important artifact instantly determines whether to put the acceptance tests in a version-control system. A focus on business-process documentation avoids overly technical specifications and keeps the specifications focused on what the system is supposed to do from a business perspective, not on test scripting. Cleaning up test code no longer requires a separate explanation. Enhancing the structure or clarity of tests is no longer something to put on the technical debt list: It's part of the standard list of tasks for delivery. The flaw in delegating the work on acceptance tests to junior developers and testers suddenly becomes obvious. The fact that useful documentation has to be well organized should prevent teams from piling up thousands of incomprehensible tests in a single directory.

By considering the living documentation a separate artifact of the delivery process, teams can also avoid overinvesting in it. They can discuss up front how much time they want to spend building the living documentation system and avoid falling into the trap of gold-plating the tests at the expense of the primary product.

I suspect that keeping specifications too abstract might be a potential pitfall of the documentation model. I expect this model to work better for software systems that are built to automate complex business processes. User-interface centric projects where the complexity isn't in the underlying processes might not benefit as much.

Remember

- There are several models of looking at Specification by Example. Different models are useful for different purposes.
- Specification by Example allows you to build up a good documentation system incrementally.
- Living documentation is an important artifact of the delivery process, as vital as code.
- Focusing on creating a business-process documentation system should help you avoid the most common long-term maintenance problems with specifications and tests.