Gojko Adzic

Specification by Example

How successful teams deliver the *right* software

MANNING

Table of Contents

18

Concluding thoughts

I began my research for this book because I was seeking external confirmation. I wanted to document that there are many teams that produce great software using agile techniques. I hoped that they were using BDD, agile acceptance testing, or what I would come to call Specification by Example. I thought I already knew how these processes worked and that I would find other people applying them in the same way I was. But the more research I did, the more unexpected lessons I learned. I found that many teams working in different contexts used a variety of practices and techniques to get to the same results. This proved that there's no such thing as a "best practice." Software development is incredibly contextual, and what might seem like a good idea for one team might be completely wrong for another.

Looking back, it surprises me how much I've learned about delivering high-quality software effectively. Some of these discoveries were completely new to me. Some were the result of viewing something with which I was familiar from a wider perspective, which gave me a much deeper understanding of the real forces behind the practices. To conclude this book, I'd like to present the top five things I've learned.

Collaboration on requirements builds trust between stakeholders and delivery team members

In *Bridging the Communication Gap*, I wrote that specification workshops have two main outputs. One is tangible: the examples or specifications. Another is intangible: a shared understanding of what needs to be done that's the result of a conversation. I stipulated that shared understanding might be even more important than the examples themselves. But it turns out the situation is much more complicated; there's another intangible output that I discovered when researching this book.

The examples of uSwitch, Sabre, Beazley, and Weyerhaeuser show that collaboration on specifications sparks a change in the culture of teams. As a result, development, analysis, and testing become better aligned and teams become better integrated.

To quote Wes Williams, after collaborating on specifications, "the trust was amazing."

Many companies I worked with use a software development model that's based on a lack of trust. Business users tell analysts what they need but don't trust them to specify it properly and require sign-off on specifications. Analysts tell developers what they need but don't trust them to deliver, so testers need to find some way to check independently that developers are honest. Because developers don't trust testers—they don't cut code—whenever testers report a problem, it's marked as impossible to reproduce, or it appears with a note like, "It works on my machine." Testers are trained not to trust anyone, almost like master spies.

A model based on mistrust creates adversarial situations and requires a lot of bureaucracy to run. Supposedly, requirements have to go through sign-off because users want to ensure what the analysts will do is right—in truth, sign-off is required so analysts can't be blamed for functional gaps later on. Because everyone needs to know what's going on, specifications go through change management; really, this ensures that nobody can be blamed for not telling others about a change. It's said that code is frozen for testing to provide testers with a more stable environment. This also guarantees that developers can't be blamed for cheating while the system was being tested. On the face of it, all these systems are in place to provide better quality. In reality, they're only alibi generators.

All these alibi generators are pure waste! By building up trust among business users, analysts, developers, and testers, we can remove the alibi generators and the bureaucracy that comes with them. Collaborating on specifications is a great way to start building up this trust.

Collaboration requires preparation

Although I stipulated that a good way to implement the process in iterations is to hold a pre-planning meeting, I didn't have anything more to say about preparing for workshops in *Bridging the Communication Gap*. I introduced the pre-planning phase because we spent too much time at the start of each workshop trying to identify important attributes for a set of examples; the real discussion started once we had something to work with. Now I see that the pre-planning meeting is a part of a much wider practice.

After talking to teams who formalized a preparation phase in different ways, I have learned that the collaboration on examples is a two-step process. In the first step, someone prepares the basic examples. In the second step, these examples are discussed with the team and extended. The goal of the preparation phase is to ensure that basic questions are answered and that there's a suggested format for examples when the team starts to discuss them. All these things can be done by a single person or two people, making the larger workshop much more effective.

For teams who worked on projects where the requirements were vague and required a lot of upfront analysis, the preparation phase started two weeks before the collaborative workshop. This allowed analysts to talk to business users, collect examples from them, and start refining the examples. Teams that had more stable requirements started working on examples a few days before, collecting the obvious open questions and addressing them. All these approaches help to run a bigger workshop more efficiently.

There are many different ways to collaborate

I suggested big, all-team workshops as the best way to collaborate on specifications in *Bridging the Communication Gap*. Again, after talking to teams in different contexts, I know that the reality is much more complex.

Many teams found that, at the start, big workshops were useful as a means to transfer the domain knowledge and align the expectations of developers, testers, and business analysts and stakeholders. But the majority of teams stopped doing big workshops after a while because they discovered that they're hard to coordinate and cost too much in terms of people's time.

Once the system is in place, trust improves, and developers and testers learn more about the domain, and smaller workshops or ad hoc conversations seem to be enough to produce good specifications. Many teams approached this from a "whoever has an interest in the story" perspective, involving only the people who would actively work on a task. When the others need to change it, they would learn about what the software does from the living documentation system.

Looking at the end goal as business process documentation is a useful model

If we think of business process documentation as the end goal of Specification by Example, many of the common automation and maintenance problems disappear. For example, the flaw in creating overly complex scripts that mimic the way the software is built becomes obvious; scripts always end up being hard to maintain and the communication value of such scripts is marginal.

As a community, we noticed this a few years ago, and many practitioners advised teams not to write acceptance tests as workflows. Although this is good advice for a majority of cases, that doesn't help when the domain is about workflows, as in processing payments. David Peterson wrote Concordion as a response to all the misuse of workflows in FIT and got a bit closer to the point by advising people to write specifications instead of scripts. Again, it's a useful rule of thumb but hard to explain to people who deal with websites. The problem is the misalignment

of models in acceptance tests or specification and the models in business;[1] one small change in the business domain has a shotgun effect on tests, which makes them hard to maintain.

If we focus on documenting business processes, the model in the specifications will be aligned with the business model and changes will be symmetric. A small change in the business domain model will result in a small change in specifications and tests. We can document business processes well before we start writing software, and they'll stay the same when we change technologies. Specifications that talk about business processes are worth much more over the long term. Business users can participate in documenting business processes and provide much better feedback than they would on acceptance tests that pertain to software.

This also tells us what to automate and how to automate it. It's easy to spot the flaws in changing specifications to include invented testing concepts or fit it into user interface interactions. If the specifications document business processes, the automation layer exercises those business processes on software. This is where the technical workflows, scripts, and simulated user interactions need to go. Automation itself isn't a goal: It's a tool to exercise the business processes.

In order to create reliable documentation, we have to validate it frequently. Automation offers one inexpensive way to do so, but it isn't necessarily the only way. Some things, such as usability, can never be properly automated; but we can still try to validate parts of specifications frequently. This addresses the problem of specifying things that are hard to automate, an issue that many teams avoid.

Long-term value comes from living documentation

Almost everyone I spoke with experienced the short-term benefits of faster deliveries and better quality. But teams who "cleaned up their tests" also got fantastic long-term benefits from them. As a consultant, I've helped many teams implement these practices, but because I don't generally stay with anyone for long, I completely miss the long-term effects. Luckily, some of the earliest adopters of these practices have now been using them for six or seven years, and they have seen great benefits in the long term as well.

Iowa Student Loan was able to change a business model quickly because they had reliable documentation. The team at ePlan Services was able to survive the absence of a key team member. The team working on the Sierra project uses "tests" as supporting documentation when they get support requests. At that point, I think it is wrong to call what they used "tests," because they don't use them for testing software: They're documentation that was built to be reliable and relevant.

Most of these teams adopted living documentation by trial and error, when they

[1] See http://dannorth.net/2011/01/31/whose-domain-is-it-anyway

were looking for easier ways to maintain tests. They restructured tests to make them more stable, aligning the models in tests and in the business. They restructured the folders containing tests to make it easier to find all the things that are relevant for a particular change, evolving a documentation system that's structured in a way that's similar to how business users think about system features.

At this point I feel relatively confident in making the bold assumption that new teams can get these benefits quicker if they intentionally create a living documentation system rather than arrive there after years of trial and error.

With that in mind, I invite you and your team to try this yourselves. After you've tried it, please share your experiences with me. You can contact me by sending an email to gojko@gojko.com.