Specification by example: key process patterns

An article from

Specification by Example
EARLY ACCESS EDITION

Gojko Adzic
MEAP Began: October 2010
Softbound Print: Spring 2011 | 275 pages
ISBN: 9781617290084

This article is taken from the book Specification by Example. The author explains the set of process patterns that facilitate change in software products to ensure that the right product is delivered efficiently.

Specification by Example is a set of process patterns that facilitate change in software products to ensure that the right product is delivered efficiently. When I say the right product, I mean software that delivers the required business effect or fulfills a business goal set by the customers or business users and it is flexible enough to be able to receive future improvements with a relatively flat cost of change. Figure 1 shows these process patterns and their relationships.

These processes are patterns because they occur in several different contexts and produce similar results. Most of the teams implemented these process ideas by trial and error, looking for better ways to do things and implement software more efficiently.

For source code, sample chapters, the Online Author Forum, and other resources, go to http://www.manning.com/adzic/
Just in time

The diagram in figure 1 shows a sequence, but successful teams do not do the entire sequence at once, for all the specifications, especially not before any development starts. They derive the scope from goals once a team is ready for more scope—for example, at the start of a project phase or a milestone. They proceed with specifications only when the team is ready to start implementing an item—for example, at the start of the relevant iteration. Do not mistake this sequence for big waterfall specifications upfront.

Deriving scope from goals

Many teams expect a customer, a product owner or a business user to come up with the scope of work before the implementation starts. Everything before that is often not considered as a problem in the domain of a software development team. Software delivery teams expect business users to specify exactly what they want, and then the teams will go and implement that. This is supposedly how the customers will be happy. In fact, this is where the issues with building the right product start. Implementation scope represents a solution to a business problem or a way to reach a business goal. By relying on customers to give us a list of user stories, use cases, or anything like that, we, in effect, ask them to design a solution. Business users are not software designers. If they define scope, the project does not benefit from the knowledge of the people in the delivery team. This often results in software that does what the customer asked for but does not deliver what they really wanted or is functionally incomplete.

Instead of requirements that are already a solution for an unknown problem, really successful teams derive scope from goals. They start with a business goal that the customer wants to achieve. They collaboratively define the scope that achieves that business goal, starting from the desired business effect. They allow the team to come up with a good solution together with the business users. The business users focus on communicating the intent of the desired feature—the value that they expect to get out of it. This helps everyone understand what is needed. The team can then suggest a solution that is cheaper, faster, easier to deliver or maintain than what the business users would come up with on their own.
Specifying collaboratively

If developers and testers are not engaged in designing the specifications, those specifications have to be separately communicated to the team. In practice, this leaves a lot of opportunities for misunderstanding and details being lost in translation. As a consequence, business users have to validate the software after delivery, and teams have to go back to change it if it fails validation. This is all unnecessary rework.

Instead of relying on any single person to get the specifications right in isolation, successful delivery teams engage in specifying the solution collaboratively with the business users. People coming from different backgrounds use different heuristics to solve problems and have different ideas. Technical experts know how the underlying infrastructure can be used better, or how emerging technologies can be applied. Testers know how the system is likely to break or where to look for potential issues, and these issues should be prevented from occurring in the first place. All this information needs to be captured as the specification of work.

Specifying Collaboratively enables us to harness the knowledge and experience of the whole team. It also creates a collective ownership of specifications, making everyone much more engaged in the delivery process.

Illustrating using examples

Natural language is ambiguous and context dependent, so any requirements described in natural language are rarely complete. They simply don’t provide a full and unambiguous context for development or testing. Developers and testers have to interpret such requirements to produce software and test scripts, and different people might interpret tricky concepts differently. This is especially problematic when something seems obvious but we need domain expertise or knowledge of a particular jargon to understand it fully. Small differences in understanding will produce a large cumulative effect, often causing rework straight after delivery. This causes unnecessary delays.

Instead of waiting for specifications to be made concrete for the first time in programming language code, successful teams make them concrete by illustrating using examples. They identify and record key examples, typical representative examples for all important classes of functions required to fulfill a specification. Developers and testers often suggest additional examples that illustrate edge cases or particularly problematic areas of the system. This flushes out functional gaps and inconsistencies and ensures that everyone involved has a shared understanding of what needs to be delivered, avoiding rework caused by misinterpretation and translation.

If the system works correctly for all the key examples, we know that it meets the specification we collaboratively agreed on. In fact, key examples effectively define what the software needs to do. They are both the target for development and an objective evaluation criterion to check whether the development is done. If the key examples are easy to understand and communicate, we can replace the requirements with the key examples.

Refining the specification

An open discussion during the collaboration on specifications builds a shared understanding of the domain, but the resulting examples often have a lot more detail than is really needed to illustrate a particular feature. Business users think about the system from the user interface perspective so they will offer examples of how something should work at the level of clicking on links and filling in fields. Using such verbose descriptions overconstraints the system and already specifies how something should be done rather than just what is to be done. Surplus details make the examples harder to communicate and understand. Key examples need to get cleaned up to be useful. By refining the specification from key examples, removing all the extraneous details, successful teams get a very concrete and precise context for development and testing. They define the target with enough detail to implement and verify it but without any additional information. They define what the software is supposed to do, not how it does it.

Refined examples can be used as an acceptance criterion for delivery. Development is not done until the system works correctly for all these examples. With some additional information to make key examples easier to understand, they make up a specification with examples, which is at the same time a specification of work, an acceptance test, and a future functional regression test.

Automating validation without changing specifications

Once we agree on specifications with examples and refine them, we can use them as a target for implementation and to validate the product. The system will be validated many times with these tests during development, to
ensure that it meets the target. Running these checks manually would introduce unnecessary delays and the feedback from that would be very slow.

Quick feedback is an essential part of developing software in short iterations or in flow mode, so we need to make the process of validating the system cheap and efficient. An obvious solution to this is automation. However, this automation is conceptually different from the usual developer or tester automation.

If we automate the validation of the key examples using traditional programming (unit) automation tools or traditional functional test automation tools, we risk introducing problems if things get lost in translation between the business specification and technical automation. Technically automated specifications will become inaccessible to business users. When the requirements change (and it is when, not if) or when developers or testers need further clarification, we won’t be able to just use the specification that we previously automated. We could keep the key examples also in a more readable form, such as Word documents or web pages but, as soon as there is more than one version of truth, we’ll have synchronization issues. That is why big paper documentation is never completely correct.

To get the most out of key examples, successful teams automate the validation without changing the information. They keep everything in the specification literally the same when automating, without any risk of mistranslation issues. As they automate validation without changing specifications, the key examples stay almost in the form they were in on a whiteboard—human-readable and accessible to all team members. If you’ve never seen a tool for automating executable specifications, this might seem unbelievable, but look at figure 2. This is a fully automated executable specification prepared as a HTML document for Concordion,¹ just one of many tools for automating executable specifications. Concordion uses special attributes in HTML documents to read inputs and expected outputs for validations and pass them on to Java or .NET automation code. The specification in figure 3 is an example of an acceptance test written for FitNesse,² another popular automation tool for executable specifications that uses values from table columns as inputs and expected outputs for automated tests.

![Figure 2 An acceptance test automated with Concordion](image)

### Free delivery

- Free delivery is offered to a **VIP** customer with **10** books in the cart.
- It is **not** offered to a **regular** customer with **10** books
- It is **not** offered to a **VIP** customer with only **9** books.

¹ [http://www.concordion.org](http://www.concordion.org)
² [http://www.fitnesse.org](http://www.fitnesse.org)

For source code, sample chapters, the Online Author Forum, and other resources, go to [http://www.manning.com/adzic/](http://www.manning.com/adzic/)
The prize pool is divided among the winners using the following distribution for winning combinations (number of correct hits out of six chosen numbers). Example below is for $2M payout pool.

<table>
<thead>
<tr>
<th>Winning Combination</th>
<th>Pool Percentage</th>
<th>Prize Pool</th>
</tr>
</thead>
<tbody>
<tr>
<td>6</td>
<td>68</td>
<td>1,360,000</td>
</tr>
<tr>
<td>5</td>
<td>10</td>
<td>200,000</td>
</tr>
<tr>
<td>4</td>
<td>10</td>
<td>200,000</td>
</tr>
<tr>
<td>3</td>
<td>12</td>
<td>240,000</td>
</tr>
</tbody>
</table>

Figure 3 An acceptance test automated with FitNesse

There are many other automation frameworks that do not require translation of key examples.

An automated specification with examples, still in a human-readable form and easily accessible to all team members, becomes an executable specification. We can use it as a target for development and easily check if the system does what we agreed. We can use that same document to get clarification from business users. If we need to change the specification, we have to do it in one place only.

**Tests are specifications, specifications are tests**

When a specification is described with very concrete examples, it can be used to test the system. After such specification is automated, it becomes an executable acceptance test.

This doesn’t mean that there are no other kinds of tests—for example, exploratory tests or usability tests fall outside of that category. The context-driven testing community tries to distinguish between these classes of tests by using the name checks for deterministic validations that can be automated and tests for non-deterministic validations that require human opinion and expert insight. With Specification by Example, testers use expert opinion and insight to design good checks in collaboration with other members of the team. Testers do not execute these checks manually, which means that they have more time for other kinds of tests.

**Validating frequently**

In order to efficiently support a software system, we have to know what it does and why. In many cases, the only way to do this is to drill down into programming code or find someone who can do that for you, if you aren’t a hieroglyphics expert. Code is often the only thing we can really trust. Most written documentation is out of date even before the project is delivered. Programmers become oracles of knowledge and bottlenecks of information.

Executable specifications can easily be validated against the system to check if it still does what it was supposed to do. If this validation is frequent, then we could have as much confidence in the executable specifications as we have in the code.

By checking all their executable specifications frequently, the teams quickly discover any differences between the system and the specifications. Because their executable specifications are easy to understand, the teams can discuss the changes with their business users and decide how to address the problems. They constantly keep their systems and executable specifications in sync.

**Evolving a documentation system**

The most successful teams are not satisfied with just a set of frequently validated executable specifications. They ensure that the specifications are well organized, easy to find and access, and consistent. As their project evolves,

---


For source code, sample chapters, the Online Author Forum, and other resources, go to http://www.manning.com/adzic/
the team’s understanding of the domain changes. Market opportunities cause changes to the business domain models. The teams that get the most out of Specification by Example update the specifications to reflect those changes, evolving a *living documentation* system.

A living documentation replaces all the artifacts that teams need for delivering the right product but in a way that fits nicely with short iterative or flow processes. André Brissette says that 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 is useful. For people that 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.

Living documentation is a reliable and authoritative source of information on system functionality, which anyone can easily access. It is as reliable as the code, but much easier to read and understand. Developers can use it to find out what the system does and why. Testers can use it for testing. Business analysts can use it as a starting point when analyzing the impact of a requested change of functionality. And it gives us regression testing for free.

Living documentation is the end-product of Specification by Example. To create a living documentation system, many teams have ended up designing a domain specific language for specifications and tests. They structured their specifications around that language to keep them consistent and easy to understand and to minimize long-term maintenance costs. When retrofitting executable specifications into already existing products, teams often had to make the system and the architecture more testable, which required senior people to plan and implement serious design changes.

If we think about executable specifications, acceptance tests and living documentation as by-products of the development process, this investment is very hard to justify. It doesn't give direct value to any of the project stakeholders. The stakeholders of living documentation, however, aren't just the customers or the business users. They are also the members of the development and maintenance teams. Many teams I interviewed have at some point ripped out the heart of their system and replaced it or rewritten the entire system while keeping their specifications with examples and using them to guide the whole effort. This is where the investment in living documentation really pays off. Such a framework is genuinely a separate product, with a different lifecycle and a different group of stakeholders.

I think that teams should consider their living documentation systems as a separate product. This is one of the things that I consider very important for the future but can't back by case studies yet. I expect that understanding that your living documentation is actually a product in its own right resolves a ton of things. For example, the question of whether to put the acceptance tests in a version control system or not goes away immediately. Cleaning up ‘test code’ no longer requires a separate explanation. Working on enhancing the structure or clarity of tests is no longer something to put on the technical debt list, it is part of a backlog for a completely separate product. The flaw in delegating the work on acceptance tests to junior developers and testers suddenly becomes obvious. Teams can discuss upfront how much time they want to invest in building this living documentation system. They won’t fall into the trap of goldplating the tests at the expense of the primary product because the tasks for the two products are not clearly separated.
Remember

• Specification by Example eliminates wasteful rework and just-in-case code.

• Key process patterns of Specification by Example are deriving scope from goals, specifying collaboratively, illustrating specifications using examples, refining the specifications, automating validation without changing the specifications, validating the system frequently, and evolving a living documentation.

• With Specification by Example, functional requirements, specifications, and acceptance tests are the same thing.

• The end result is a living documentation system that explains what the system does; it is as relevant and reliable as the programming language code but much easier to understand.

• Teams in different contexts use different practices to implement these process patterns.