Gojko Adzic

Specification by Example

How successful teams deliver the *right* software

MANNING

Table of Contents

5

Deriving scope from goals

The F-16 Fighting Falcon is arguably the most successful jet fighter ever designed. This is all the more remarkable because it succeeded against all odds. In the 70s, when the F-16 was designed, jet fighters had to be built for speed; range, weaponry, and maneuverability were of little importance to get a production contract.[1] Yet it was the range and maneuverability of the F-16 that made it ideal for its role in combat and ensured its success.

In *97 Things Every Architect Should Know*,[2] Einar Landre quotes Harry Hillaker, the lead designer of the F-16, saying that the original requirement for the aircraft was that it reach speeds of Mach 2-2.5. When Hillaker asked the U.S. Air Force why that was important, they responded that the jet had to "to be able to escape from combat." Although Hillaker's design never got above Mach 2, it allowed pilots to escape from combat with superior agility. It featured many innovations, including a frameless bubble canopy for better visibility, a reclined seat to reduce the effect of g-forces on the pilot, a display that projects combat information in front of the pilot without obstructing his view, and side-mounted control sticks to improve maneuverability at high speed. With these features, the F-16 was superior to alternative designs—and less expensive to produce. It won the design competition. More than 30 years later, it's still in production. With more than 4,400 aircraft sold to 25 countries,[3] the model is a great commercial success. It's also one of the most popular fighter jets and is often featured in action films, such as *X2* and *Transformers: Revenge of the Fallen.*

The F-16 was successful because the design provided a better and cheaper solution than what the customer asked for. The original requirements, including the demand for Mach 2.5 speed, formed one possible solution to a problem—but this problem wasn't

[1] See Kev Darling's book *F-16 Fighting Falcon (Combat Legend)* (Crowood Press, 2005).
[2] Richard Monson-Haefel, *97 Things Every Software Architect Should Know* (O'Reilly Media, 2009).
[3] See http://www.lockheedmartin.com/products/f16

Chapter 5. Deriving scope from goals. Specification by Example: How successful teams deliver the right software, ISBN: 9781617290084

effectively communicated. Instead of implementing the requirements, the designers sought a greater understanding of the problem. Once they had it, they could pinpoint the real goals and derive their design from those, rather than from suggested solutions or arbitrary expectations about functionality. That's the essence of successful product design, and it's just as important in software design as in aircraft development.

Most of the business users and customers I work with are inclined to present requirements as solutions; rarely do they discuss goals they want to achieve or the specific nature of problems that need solutions. I've seen far too many teams suffer from the hazardous misconception that the customers are always right and that what they ask for is set in stone; this leads teams to blindly accept suggested solutions and then struggle to implement them. Successful teams don't do this.

Like the F-16 designers, successful teams push back for more information about the real problem and then collaborate to design the solution. They do this even for scope. Scope implies a solution. Instead of passing the responsibility for defining scope onto someone else, successful teams are proactive and collaborate to determine good scope with the business users, so that their goals are met. This is the essence of deriving scope from goals.

Collaborating on deriving scope from goals is undoubtedly the most controversial topic in this book. In the last five years, the surge in popularity of value chains in software development has increased awareness of the idea of collaborating on scope and deriving it from business goals. On the other hand, most teams I work with still think that project scope isn't under their control and expect customers or business users to fully define it. In the course of my research for this book, I found a pattern of teams deriving their project scope from goals collaboratively—but this practice is much less common than other key patterns.

I originally thought about leaving this chapter out. I decided to include it for three reasons:

- Defining scope plays an important role in the process of building the right software. If you get the scope wrong, the rest is just painting the corpse.
- In the future, this will be one of most important software development topics, and I want to raise awareness about it.
- Defining scope fits nicely into designing processes from value chains, which are becoming increasingly popular because of lean software development.

In the following two sections I present techniques for influencing scope for teams that have direct control over it and for teams that don't. Teams that have high-level control of their project scope can be proactive and begin to build the right scope immediately. Unfortunately, many teams in several of the large organizations I work with don't have that kind of control—but this doesn't mean they can't influence scope.

Building the right scope

Use cases, user stories, or backlog items provide a broad definition of a project's scope. Many teams consider such artifacts to be the responsibility of the business users, product owners, or customers. Asking business users to provide the scope is, in effect, relying on individuals who have no experience with designing software to give us a high-level solution. Designing a solution is one of the most challenging and most vital steps. Now is the time for a mandatory Fred Brooks quote: In *The Mythical Man-Month*[4] he wrote, "The hardest single part of building a software system is deciding precisely what to build." Albert Einstein himself said that "the formulation of a problem is often more essential than its solution."

Currently, user stories are the most popular way to define the scope for agile and lean projects. User stories did a fantastic job of raising awareness of business value in software projects. Instead of asking business users to choose between developing an integration platform and building transaction CRUD (Create Update Delete) screens, user stories allowed us to finally start talking to them about things that they could understand and reasonably prioritize. It's important to note that each story should have a clearly associated business value. Business users often choose that value statement arbitrarily (and it's usually the tip of the iceberg). But when we know what a story is supposed to deliver, we can investigate that further and suggest an alternative solution. Christian Hassa of TechTalk explains:

> People tell you what they think they need, and by asking them "Why" you can identify new implicit goals they have. Many organizations aren't able to specify their business goals explicitly. However, once you derived the goals, you should again reverse and derive scope from the identified goals, potentially discarding the originally assumed scope.

This is the essence of a practice I call *challenging requirements* in *Bridging the Communication Gap*. I still think that challenging requirements is an important practice, but doing so is reactive. Although that's definitely better than passive—which best describes the way most teams I've seen work with scope—there are emerging techniques and practices that allow teams to be much more proactive in achieving business goals. Instead of reacting to wrong stories, we can work together with our business users on coming up with the right stories in the first place. The key idea is to start not with user stories but with business goals and derive scope from there collaboratively.

[4] Fred Brooks, *The Mythical Man-Month: Essays on Software Engineering* (Addison-Wesley, 1975).

Understand the "why" and "who"

User stories generally have three parts: "As a __ I want __ in order to __ ." Alternative formats exist, but all have these three components.

Understanding why something is needed and who needs it is crucial to evaluating a suggested solution.

The same questions can be applied to project scope on a much higher level. In fact, answering those questions at a higher level can push a project into a completely different direction.

Peter Janssens from iLean in Belgium worked on a project where he was on the other end—he was the person giving requirements in the form of solutions. He was responsible for an application that stored local traffic sign information. It started as a simple Access database for Belgium but quickly grew to cover most of the countries in the world. The company had a data collector in each country, and they were all using local Access databases, which were occasionally merged.

To make the work more efficient and prevent merging problems, they decided to take the database online and make a web application to maintain it. They spent four months contacting suppliers and comparing bids before finally selecting one offer. The estimated cost for this application was 100,000 euros. But the project took a completely different turn once they thought about who needed to use the application and why. Janssens says:

> The day before the go/no a guy from engineering asked again for the problem, to understand it better. I said, "We need a web solution for the central database." He said, "No, no, let's not jump to conclusions. Don't elaborate immediately which solution you want, please explain it to me." I explained again. And then he said, "So, your problem is actually that you need a single source to work on because you are losing time on merging." "Yes," I said, "correct."
>
> He had a second question: "Who is working on it?" I said, "Look, at this moment we have 10 groups of countries, so 10 people." We had a look at the databases and understood that this type of traffic information doesn't change that often, maybe once or twice a year per country. He said, "Look Peter, your problem will be solved by tomorrow." By tomorrow he added the database to their Citrix (remote desktop) server.

The application had to support ten users in total, and they were using it only to update traffic sign information, which changed infrequently. The Access application could cope with the volume of data; the only real problem they had was merging. Once a technical engineer understood the underlying problem, he could offer a much cheaper solution than the one originally suggested. Janssens explains:

> What I learned is that this was a real confrontation situation—it's always important to understand the core problem that leads to a request. So understanding "why" is important. Eventually, the Citrix solution came to him when we talked about the "who" question. On average there was one user a month working on it.

Even at a scope level, the solution is already implied. Without going into the possible user stories or use cases and discussing specifications for tasks, the fact that someone suggested a web application implies a solution. Instead of spending five months selecting a supplier and even longer to deliver the project, they solved the problem with a quick fix that cost nothing. This is an extreme case, but it demonstrates that understanding why someone needs a particular application and how they'll use it often leads to better solutions.

Understand where the value is coming from

In addition to helping us design a better solution, understanding where the value comes from is immensely helpful when it comes to prioritization. Rob Park's team at a large U.S. insurance provider looks at prioritization only from a higher feature level, which keeps them from going through the same process on the lower story level and saves a lot of time. Park says:

> We keep things at a high level, describe the business value and what's really core to the feature. We break the feature down into stories, which we try to make as small as possible. An example of a feature would be: Deliver proof of insurance as PDF for 14 states. The big thing I've been trying to push, especially from the business side, is "how much is this worth, put the dollar value behind it." In that particular case we were able to get one of the senior guys to say, "Well, 50% of the calls were for this and 50% of those calls were for the proof of insurance cards, so 25% of the calls were dealing with that." They know how many calls they have and how much they would be able to save by generating this PDF instead of having to do all the copy and paste stuff that they were doing before, so they were actually able to put some numbers behind this, which was really cool.

Raising the discussion to the level of the goals allows teams to deal with scope and priorities more efficiently than just doing so at the story level.

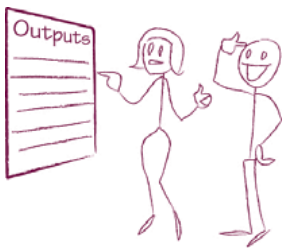
One good example of an area where this helps is effort estimation. Rob Park's team found that discussing goals enabled them to stop wasting time on estimating effort for individual stories:

> We don't really want to bother with estimating stories. If you start estimating stories, with Fibonacci numbers for example, you soon realize that anything eight or higher is too big to deliver in an iteration, so we'll make it one, two, three, and five. Then you go to the next level and say five is really big. Now that everything is one, two, and three, they're now really the same thing. We can just break that down into stories of that size and forget about that part of estimating, and then just measure the cycle time to when it is actually delivered.

In *Software by Numbers*,[5] Mark Denne and Jane Cleland-Huang describe a formal method for prioritization that's driven by business value by dividing scope into Minimum Marketable Features. From my experience, predicting how much money something is going to earn is as difficult and prone to error as predicting how long it's going to take to implement that feature. But if your domain allows you to put numbers on features, this will help get the business users involved. Asking them to prioritize features or even business goals works better than asking them to prioritize low-level stories or tasks.

Understand what outputs the business users expect

When goals are hard to pin down, a useful place to start is the expected outputs of the system: Investigate why they're needed and how the software can provide them. Once you nail down expected outputs, you can focus your work on fulfilling the requirements that come with them. Analyzing why those outputs are required leads to formulating the goals of the project.



Instead of trying to collaborate with business users on specifying how to put things into the system, we should start with examples of outputs. This helps engage business users in the discussion and gives them a clear picture of what they'll get out of the system.

[5] Mark Denne and Jane Cleland-Huang, *Software by Numbers: Low-Risk, High-Return Development* (Prentice Hall, 2003).

Wes Williams worked on a project at Sabre where a delay in building the user interface caused a lot of rework:

> The acceptance tests were written against a domain [application layer], before our customer could see the GUI. The UI was delayed for about four months. The customer thought completely differently about the application when they saw the UI. When we started writing tests for the UI they had much more in them than the ones written for the domain [layer]. So the domain code had to be changed, but the customer assumed that that part was done. They had their test there, they drove it and it was passing, and they assumed it was done.

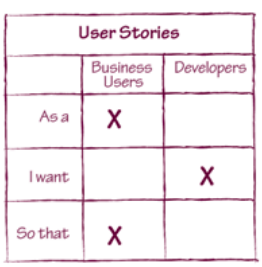
Expected outputs of a system help us discover the goals and determine exactly what needs to be built to support them. Adam Geras used this idea to focus on building the right thing even before agile projects:

> We've used something that we call "report-first" on many of our projects, but it is at the epic story level and our experience with it is mostly in the ERP implementation space. Not agile projects. It has served us extremely well because the rework required to find that one data element that was missing on a report can be extensive. We've been able to avoid that rework by thinking about the outputs first.

Starting with the outputs of a system to derive scope is an idea from the BDD community. This idea has been getting a lot of attention recently because it eliminates a common problem. On many of my early projects, we focused on process flow and putting the data into the system initially. We left the end results of processes, such as reports, for later. The problem with this approach is that the business users can become engaged only when they see results at the visible output stage, which often causes rework. Working from the outputs ensures that the business users can always provide feedback.

Have developers provide the "I want" part of user stories

When: Business users trust the development team



The team at uSwitch collaborates with their business users to define user stories. The business users specify the parts of the story that name a stakeholder and an expected benefit, and the development team specifies the part that implies a solution. In the standard user story format, this would mean that the business users provide direction for the "as a __" and "in order to __" statements, whereas the developers provide content for the "I want __" statement.

A great way to obtain the right scope for a goal is to firmly place the responsibility for a solution on the development team.

If you are fortunate enough to have high-level control of project scope, make sure to involve developers and testers in the discussions about it and focus the suggested solutions on fulfilling clearly defined business goals. This eliminates a lot of unnecessary work later on and sets the stage for better collaboration on specifications.

Parts of a user story

A *user story* describes how a user will get a specific value out of a system. User stories are commonly used by teams to plan and prioritize the scope of short-term work. They're often defined in three parts:

- As a *stakeholder*
- In order to *achieve something valuable*
- I want *some system function*

For example, "As a marketing manager, so that I can market products directly to customers, I want the system to request and record personal information when customers register for a loyalty program."

Different authors suggest different ordering and phrasing of these three parts, but all agree that those three need to be captured. For the purposes of this book, variations in ordering or naming in parts of a user story are irrelevant.

Collaborating on scope without high-level control

For most teams I work with, especially those in big companies, scope is something passed to them from a higher instance. Many teams think that it's impossible to argue about business goals when they maintain only a piece of a large system. Even in those situations, understanding what the business users are trying to achieve can help you focus the project on things that really matter.

Here are some tips for effectively collaborating on project scope when you don't have a high-level control of the project.

Ask how something would be useful

Stuart Ervine worked on a back-office application for a large bank that allowed business users to manage their counterparty relationships in a tree-like hierarchy—a perfect example of a small piece of a large system. Even then, they were able to push back on tasks and get to the real requirements.

Ervine's team was tasked with improving the performance of the hierarchy, which sounds like a genuine business requirement with a clear benefit. But the team could not replicate any performance issues on their part, so any serious improvements would require infrastructural changes.

They asked the users to tell them how improved performance would be useful. It turned out that the business users had been manually performing a complex calculation by going through the hierarchy and adding account balances. They had to open and close tree branches in the user interface for a large number of counterparties and add account balances—a slow and error-prone calculation process.

Instead of improving the performance of the hierarchy, the team automated that calculation for the business users. This made the calculation almost instantaneous and significantly reduced the possibility of errors. This solution delivered better results and was cheaper than the one originally requested.

Instead of a technical feature specification, we should ask for a high-level example of how a feature would be useful. This will point us towards the real problem.

In *Bridging the Communication Gap,* I advise asking why and repeating the question until the answer starts mentioning money. I now think that asking for an example of how a feature will be useful is a much better way to get to the same result. Asking why something is needed can sound like a challenge and might put the other person in a defensive position, especially in larger organizations. Asking how something would be useful starts a discussion without challenging anyone's authority.

Ask for an alternative solution

In addition to asking for an example of how something would be useful, Christian Hassa advises discussing an alternative solution to get to the real business goals. Hassa explains:

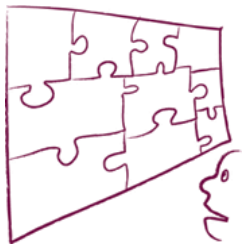
> Sometimes people still struggle with explaining what the value of a given feature would be (even when asking them for an example). As a further step, I ask them to give an example and say what they would need to do differently (work around) if the system would not provide this feature. Usually this helps them then to express the value of a given feature.

A good strategy for discovering additional options from a business perspective is to ask for an alternative solution.

Asking for alternative solutions can make whoever is asking for a feature think twice about whether the proposed solution is the best one. It should also start a discussion about alternatives with the delivery team.

Don't look only at the lowest level

Many teams, influenced by the need to slim down delivery items so that they can fit into an iteration, now break down backlog items to a low level. Although this helps streamline process flow, it might cause the team to lose sight of the big picture.



As a process, Specification by Example works both for high-level and lower-level stories. Once we have a high-level example of how something would be useful, we can capture that as a high-level specification. Such high-level examples allow us to objectively measure whether we've delivered a feature.

Ismo Aro worked on a project at Nokia Siemens Networks where his team experienced setbacks because they didn't have higher-level specifications. He says:

> User stories have to fit into the sprint. When there is a bunch of those that are done, they're tested in isolation. The larger user story isn't actually tested. When the user stories are small grained you can't really tell from the backlog whether things are really done.

Splitting larger user stories into smaller ones that can be delivered individually is good practice. Looking at higher-level stories is still required in order to know when we're done. Instead of a flat, linear backlog, we need a hierarchical backlog to look at both levels.

Lower-level specifications and tests will tell us that we've delivered the correct logic in parts; a higher-level acceptance test will tell us that all those parts work together as expected.

Make sure teams deliver complete features

When: Large multisite projects

Wes Williams blamed the division of work for the problem described in the section "Understand what outputs the business users expect." Teams delivered components of the system (in this case the domain layer and the user interface), which made it hard to divide the work so that they could discuss the expected output for each team with their customers. So they reorganized work into teams that could deliver complete features. Williams commented:

> It took us about six months to put it to feature teams. This made a big difference especially in the sense that it removed some duplication, a lot of repeating, and a lot of rework. Fortunately, we already had a lot of tests that helped us do this. We did have to go back and add features but it was mostly adding—not changing.

When teams deliver features end-to-end, they can get more thoroughly engaged with business users in designing scope and determining what needs to be built, simply because they can discuss full features with the users. For more information on feature teams, see the *Feature Team Primer*.[6]

Even without high-level control of project scope, teams can still influence what gets built by:

- Actively challenging requirements
- Understanding the real business goals
- Understanding who needs what functionality and why

The result will not be as effective as it would be if the right scope had been derived from business goals from the start. But this approach prevents unnecessary rework later in the process and ensures that the business users get what they need.

Further information

At the moment there's a lot of innovation in this field. True to the nature of this book, I've only written about techniques utilized by the teams I interviewed.

[6] http://www.featureteams.org

Emerging techniques deserve to be written about, but that's material for another book. To learn more about cutting-edge techniques for deriving scope from goals and mapping out the relationship between them, look for resources on these topics:

- *Feature injection*—A technique to iteratively derive scope from goals through high-level examples
- *Effect mapping*—A visualization technique for project scope through hierarchical analysis of goals, stakeholders, and features
- *User story mapping*—A hierarchical mapping technique for user stories that provides a "big picture" view

Unfortunately, there's little published material on any of the emerging practices. As far as I know, the only published work about feature injection is a comic[7] and the next best thing is a set of scans from Chris Matts's notebook on Picasa.[8] The only published material on effect maps is a book in Swedish with a poor English translation called *Effect Managing IT*[9] and a whitepaper I published online.[10] Jeff Patton features a lot of great material about passive and reactive scoping problems on his blog,[11] and he's been writing a book on agile product design, which I hope will offer more coverage of this field.

Remember

- When you're given requirements as tasks, push back: Get the information you need to understand the real problem; then collaboratively design the solution.
- If you can't avoid getting tasks, ask for high-level examples of how they would be useful—this will help you understand who needs them and why, so you can then design the solution.
- To derive the appropriate scope, think about the business goal of a milestone and the stakeholders who can contribute or be affected by that milestone.
- Start with the outputs of a system to get the business users more engaged.
- Reorganize component teams into teams that can deliver complete features.
- Investigate emerging techniques, including feature injection, user story mapping, and effect mapping to derive scope from goals effectively.

[7] See www.lulu.com/product/file-download/real-options-at-agile-2009/5949486 to download a free copy.
[8] http://picasaweb.google.co.uk/chris.matts/FeatureInjection#
[9] Mijo Balic and Ingrid Ottersten, *Effect Managing IT* (Copenhagen Business School Press, 2007).
[10] http://gojko.net/effect-map
[11] www.agileproductdesign.com