Gojko Adzic

Specification by Example

How successful teams deliver the *right* software

MANNING

Table of Contents

7

Illustrating using examples

Examples are a good way to avoid ambiguities and communicate with precision. We use examples in everyday conversation and in writing without even thinking about it—when I searched online for the phrase "for example," Google returned more than 210 million pages that use this term.

With traditional specifications, examples appear and disappear several times in the software development process. Business analysts often get examples of existing orders, invoices, and reports from business users, which they translate into abstract requirements. Developers invent examples to explain edge cases and clarify them with business users or analysts and then translate the cases to code, without recording the examples. Testers design test cases that are examples of how the system is expected to work; they keep these examples to themselves and don't communicate them to programmers or analysts.

Everyone invents their own examples, but there's nothing to ensure that these examples are even consistent, let alone complete. In software development, this is why the end result is often different from what was expected at the beginning. To avoid this, we have to prevent misinterpretation between different roles and maintain one source of truth.

Examples are a good tool for avoiding communication problems. We can avoid playing the telephone game by ensuring that we capture all the examples—from start to finish—and use them consistently in analysis, development, and testing.

Marta Gonzalez Ferrero was working as a test lead at Beazley when they introduced Specification by Example. According to her, the development team was committing to more work than they could produce, and they often realized they needed a lot more information than they were getting at the start of the implementation. The situation was further complicated by the fact that they were running six-week iterations, and the development team and the business analysts were on different continents. The acceptance criteria that the programmers were receiving from the business analysts was relatively abstract (for example, "make sure that for this business

unit all correct products are displayed"). Finding out that something important was missing halfway through an iteration would seriously disrupt the output. One iteration ended with customers saying that the team delivered something completely different from what was expected. The last week of each iteration was reserved for the model office: effectively, an iteration demonstration. Ferrero traveled to the United States for one model office and worked with business analysts on illustrating requirements with examples for two days. As a result, the team committed to 20% less work for the next iteration and delivered what they promised.

"The feeling in the team was also much better," said Ferrero. "Before that, [the developers] were working with a feeling that they were making it up as they go, and had to wait for feedback from business analysts." According to Ferrero, the amount of rework dropped significantly after they started illustrating requirements using examples.

Ferrero's wasn't the only team to experience results like these. Almost all the teams profiled in this book confirmed that illustrating requirements using examples is a much more effective technique than specifying with abstract statements. Because examples are concrete and unambiguous, they're an ideal tool for making requirements precise—this is why we use them to clarify meaning in everyday communication.

In *Exploring Requirements,*[1] Gerald Weinberg and Donald Gause write that one of the best ways to check if requirements are complete is to try designing black-box test cases against them. If we don't have enough information to design good test cases, we definitely don't have enough information to build the system. Illustrating requirements using examples is a way to specify how we expect the system to work with enough detail that we can check that assertion. Examples used to illustrate requirements are good black-box tests.

From my experience, it takes far less time to illustrate requirements with examples than to implement them. Concluding that we don't have enough information to illustrate something with examples takes far less time than coming to the same realization after trying to implement the software. Instead of starting to develop an incomplete story only to see it blow up in the middle of an iteration, we can flush such problems out during the collaboration on specifications while we can still address them—and when the business users are still available.

In May 2009 I ran a three-hour workshop on Specification by Example[2] during the Progressive .NET tutorials. Around 50 people, mostly software developers and testers, participated in this workshop. We simulated a common situation: A customer directs the team to a competitor site and asks them to copy some functionality.

[1] Gerald M. Weinberg and Donald C. Gause, *Exploring Requirements: Quality Before Design* (New York, Dorset House, 1989).

[2] See http://gojko.net/2009/05/12/examples-make-it-easy-to-spot-inconsistencies

I copied the rules of a blackjack game from a popular website and asked the participants to illustrate those rules using examples. Although the requirements were taken from a real website and fit on a single sheet of paper, they were ambiguous, redundant, and incomplete. In my experience, this is often the case when requirements are captured as Word documents.

The participants were divided into seven teams, and each team had only one person with knowledge of blackjack. After the workshop, all the participants agreed that discussing realistic examples helped flush out inconsistencies and functional gaps. By running a feedback exercise (see sidebar), I measured the level of shared understanding. Six out of seven teams came up with the same answers to difficult edge cases, even though most people on the team had no previous exposure to the target domain. Illustrating requirements with examples is a very effective way to communicate domain knowledge and ensure a shared understanding. I've seen this effect on real software projects, as have many teams that I interviewed for this book.

Feedback exercises

Feedback exercises are a good way to check whether a group of people has a shared understanding of a specification. When someone suggests a special case after a story has been discussed, the person running the workshop should ask the participants to write down how they think the system should work. The entire group then compares the answers. If they all match, everyone understands the specification in the same way. If the answers don't match, then it's useful to organize the results into clusters and get one person from each cluster to explain their answers. The discussion will reveal the source of misunderstanding.

Illustrating requirements using examples is a simple idea, but it's far from easy to implement. Finding the right set of examples to illustrate a requirement turns out to be quite a challenge.

In this chapter, I begin by putting things in perspective using an example of the process. Then, I present good ideas for identifying the right set of examples to illustrate a business function. Finally, I cover ideas for illustrating cross-cutting functionality and concepts that aren't easy to capture with precise values.

Illustrating using examples: an example

To clarify how illustrating a requirement using examples works, let's take a look at an example involving a fictional company, ACME OnlineShop. This is the only fictional company in the book, but I had to invent one to keep the example simple. Acme is a small web store whose development team started a Specification workshop. Barbara, a

business analyst, spent some time the week before with Owen, the company owner, to get some initial examples. She's facilitating the workshop and introduces the first story:

Barbara: The next thing on the list is free delivery. We have arranged deal with Manning to offer free delivery on their books. The basic example is this: If a user purchases a Manning book, say *Specification by Example*, the shopping cart will offer free delivery. Any questions?

David, a developer, spots a potential functional gap. He asks: Is this free delivery to anywhere? What if a customer lives on an island off South America? That free delivery will cost us much more than we earn from the books.

Barbara: No, this isn't worldwide, just domestic.

Tessa, a tester, asks for another example. She says: The first thing I'd check when this comes for testing is that we don't offer free delivery for all books. Can we add one more case to show that the free delivery is offered only for Manning books?

Barbara: Sure. For example, *Agile Testing* was published by Addison-Wesley. If a user buys that, then the shopping cart won't offer free delivery. I think this is relatively simple; there isn't a lot more to it. Can anyone think of any other example? Can we play around with the data to make it invalid?

David: There aren't any numerical boundary conditions, but we could play with the list in the shopping cart. For example, what happens if I buy both *Agile Testing* and *Specification by Example*?

Barbara: You get free delivery for both books. As long as a Manning book is in the shopping cart, you get free delivery.

David: I see. But what if I buy *Specification by Example* and a fridge? That delivery would be much more expensive than our earnings from the book.

Barbara: That might be a problem. I didn't talk about that with Owen. I'll have to get back to you on this. Any other concerns?

David: Not apart from that.

Barbara: OK. Do we have enough information to start working, apart from the fridge problem?

David and Tessa: Yes.

Barbara: Great. I'll get back to you on that fridge problem early next week.

Examples should be precise

Good examples help us avoid ambiguities. In order to do that, there must be no room for misunderstanding. Each example should clearly define the context and how the system should work in a given case and, ideally, describe something we can easily check.

Don't have yes/no answers in your examples

When: The underlying concept isn't separately defined

When describing processes, many teams I interviewed oversimplified examples by using yes/no answers. This can be misleading and give people the false sense that they have shared understanding when they don't.

For example, TechTalk had this issue when illustrating the requirements for email alerts in a web-based refund system. They had examples of conditions about when to send emails, but they didn't discuss the email contents. "The customer expected us to include the failing case and the resolution, and we didn't capture that," said Gaspar Nagy, a developer who worked on this system.

I ran a specification workshop for a major investment bank. The team was discussing how payments are routed to different systems. They started by listing examples in a table with conditions on the left and different subsystems on the right, marking the columns yes or no depending on whether the destination receives a transaction or not. Instead of yes/no, I asked them to write down the key attributes of the messages sent to each of the systems. At that point, several interesting cases came up that most of the developers misunderstood. For example, instead of a transaction update, one of the systems was expecting two messages: one to cancel an existing transaction and one to book a new one.

Watch out for examples that have yes/no answers and try to rewrite them to be more precise.

You can still leave yes/no in examples as long as the underlying concept is illustrated separately. For example, one set of examples can tell you whether an email is sent or not, while another set of examples illustrates the email's contents.

Avoid using abstract classes of equivalence

When: You can specify a concrete example

Classes of equivalence (such as "less than 10") or variables can create an illusion of shared understanding. Without choosing a concrete example, different people might, for example, be unclear on whether negative values are included or left out.

When equivalence classes are used as input parameters, expected outputs have to be specified as formulas with variables representing the input values. This effectively replicates the description of the functionality. It doesn't provide a concrete example to verify it; the value of illustrating using examples is lost.

Classes of values have to be translated into something concrete for automation, which means that whoever automates the validations will have to translate the specifications into automation code. This means more opportunities for misunderstanding and misinterpretation.

From my experience, the things that seem obvious in requirements can trick us the most. Confusing concepts are discussed and explored. But the ones that seem clear—with different people understanding them differently—will go undetected and cause problems.

Instead of classes of equivalence, always use a representative concrete example. Concrete examples allow us to automate the validation of specifications without changing them and ensure that all team members have a shared understanding.

You can safely use equivalence classes as expected outputs, particularly when the process you're trying to describe isn't deterministic. For example, stating that the result of an operation should be between 0.1 and 0.2 still makes a specification testable. A concrete value makes it more precise if the process is deterministic; try to use concrete values, even for outputs.

Examples should be complete

We should have enough examples to describe the entire scope of a feature. Expected behavior in primary business cases and simple examples are a good start, but they're rarely the sum of what needs to be implemented. Here are some ideas on how to extend an initial set of examples to provide a full picture of functionality.

Experiment with data

Once you have a set of examples that you think is complete, look at the structure of the examples and try to come up with valid combinations of inputs that could violate the rule. This helps reveal what you might have missed, making the specification more complete and stronger.

If the examples include numerical values, try to use large and small numbers around different boundary conditions. Try to use a zero or negative numbers. If the examples include entities, consider whether you can use more than one object, whether an example without that entity is still valid, and what happens if the same entity is specified twice.

When collaborating on specifications, I expect testers in particular to help with finding examples like these. They should have techniques and further heuristics to identify potential problematic cases.

Many of the technical edge cases you identify won't represent valid examples; that's fine. Don't cover them in detail unless you're demonstrating error messages for invalid arguments (in which case these are valid examples for that business function). Thinking about these different cases might flush out inconsistencies and edge cases you might not have thought about earlier.

One risk of experimenting with data is that the output will have too many examples with insignificant differences. This is why the next step, refining the specification (described in the next chapter), is important.

Ask for an alternative way to check the functionality

When: Complex/legacy infrastructures

In complex IT systems, it's easy to forget about all the places where you should send a piece of information.

To test whether you have a good set of examples specifying a story, ask the business users to think of an alternative way to verify the implementation.

"How else would you be able to test this?" is a good question to kick off that discussion. Bas Vodde also suggests asking, "Is there anything else that would happen?" When I asked this question in the same specification workshop discussed in "Don't have yes/no answers in your examples," we discovered a legacy data warehouse that some people thought should receive the transaction and that others thought should be ignored. This discovery prompted us to have a discussion and close this functional gap.

Pascal Mestdach had similar experiences on the Central Patient Administration project at IHC. They often had problems when customers presumed data stored in a new application would also be sent to the legacy application during the migration period, but the team didn't understand these requirements. Asking the customers for an alternative way to test the feature would have revealed their expectation that they see the information in the legacy system as well.

Asking for an alternative way to check functionality is also a useful way to help the team discuss the best place to automate the validation.

Examples should be realistic

Ambiguities and inconsistencies are flushed out when we illustrate a feature with examples, because examples focus the discussion on real cases instead of abstract rules. For this to work, the examples have to be realistic. Invented, simplified, or abstracted examples won't have enough detail or exhibit enough variation for this. Watch out for abstract entities, such as "customer A." Find a real customer who has the characteristic you want to illustrate, or, even better, focus on the characteristic and not the customer.

Avoid making up your own data

When: Data-driven projects

Using real data is important on data-driven projects, when a great deal can depend on slight variations and inconsistencies.

Mike Vogel from Knowledgent Group worked on a greenfield project using metadata-driven ETL to populate a data repository for pharmaceutical research. They used Specification by Example, but both the team and the customer invented examples to illustrate the functionality instead of looking at real data samples. He says the approach didn't help them avoid inconsistencies:

> They [customer representatives] were making up examples; they didn't deal with real variations. They assumed they could do certain things and left it out of examples. When the data from real systems came in, there were always too many surprises.

This is an even greater problem with projects that involve legacy systems, because legacy data often defies expected consistency rules (and rules of logic in general).

Jonas Bandi worked at TechTalk on rewriting a legacy application for school data management where significant complexity resulted from understanding existing legacy data structures and relationships. They expected that Specification by Example would protect them from boomerangs (see "Watch out for boomerangs" in chapter 4) and bugs, but this didn't happen. They were inventing examples based on their understanding of the domain. The real legacy data often had exceptions that surprised them. Bandi says:

> Even when the scenarios [test results] were green and everything looked good, we still had a lot of bugs because of the data from the legacy application.

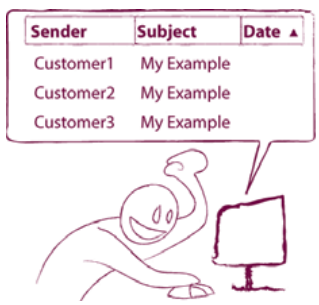
To reduce the risk of legacy data surprising the team late in the iteration, try to use realistic data from the existing legacy system in the examples instead of specifying completely new cases.

Using existing data might require some automated obfuscation of sensitive information, and it has an impact on data management strategies for automation. For some good solutions to this problem, see "Test data management" in chapter 9.

Get basic examples directly from customers

When: Working with enterprise customers

Teams that sell enterprise software to several customers rarely have the luxury of involving customer representatives in collaborative specification workshops. Product managers collect requirements from different customers and decide on release plans. This introduces the possibility of ambiguity and misunderstanding. We can have perfectly precise and clear examples that don't capture what the customers want.



Ensure that the examples used to illustrate the specifications are realistic. Realistic examples contain data that comes from the clients.

We can apply the same trick used to ensure shared understanding inside the team when we work with external stakeholders. André Brissette uses customer emails as a starting point for the discussion about automated dialogs in the Talia system:

> They would write an email such as, "It would be easier if I could ask this to Talia, and she would tell me this, and then I would be able to do that." In this case, the user provides the first draft of the dialog.

Brissette records emails like these and uses them as the initial examples to illustrate the required features. This ensures that the external stakeholders' requests are satisfied. See figure 7.1 for an example of the resulting specification. Note that this example should ideally be further refined later. See the section "Scripts are not specifications" in chapter 8.

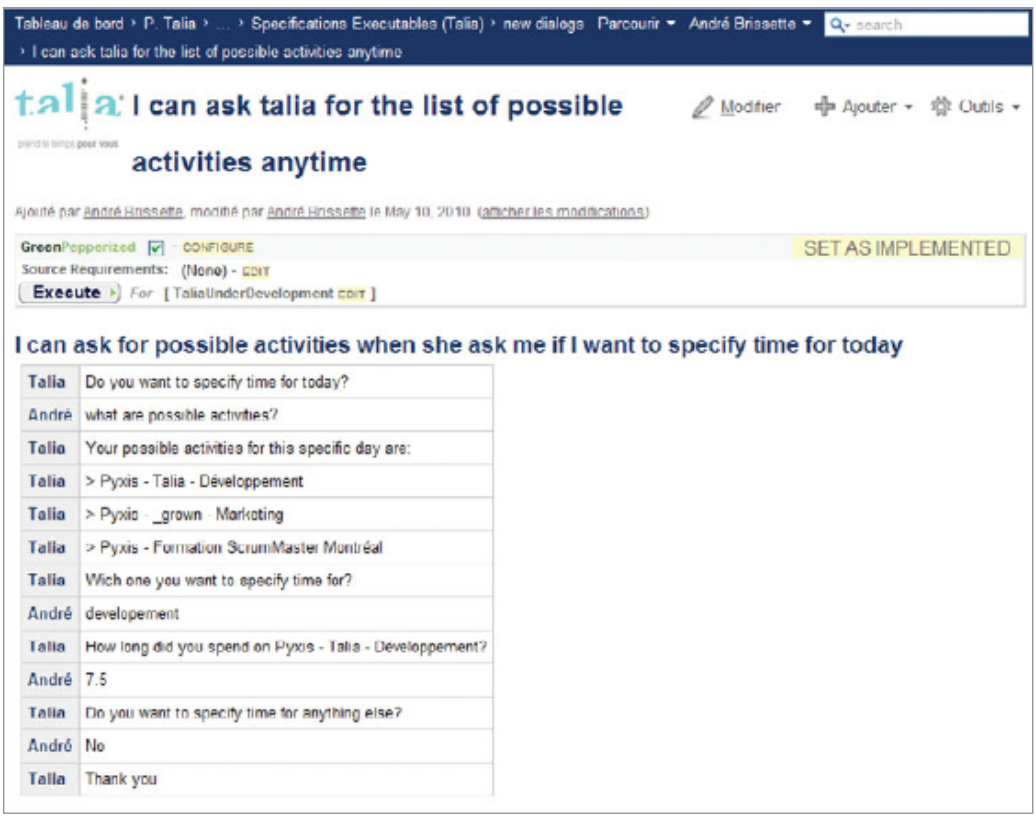


Figure 7.1 Example of a customer dialog used as a specification for the Talia system

Adam Knight's team at RainStor uses this approach to develop an archiving system for structured data. They work with customers to get realistic data sets and expected targets for representative queries. When the customer can't give them a specific use case, they push back and ask for examples, sometimes organizing workshops with the customers. A common example when customers can't give them a specific use case is when a reseller who doesn't yet have a buyer wants the system to support something because they suspect it will make it easier to sell. One example is a request to mirror functionality available in email archiving systems. Knight says:

> They looked at an email archiving system and said we need to be able to work in the same way. An email archiving system would have thousands of emails, but in our system you could have billions of records. Do you want the same level of granularity? What about logging? That is the most difficult kind of requirement. Generally we try to push back and get examples. We arrange demos to prototype functionality and walk through that.

To avoid ambiguities and misunderstanding between what the product manager thinks the customers need and what they ask for, insist on examples when communicating with customers. These examples can then be used to kick-start the discussion during specification workshops. They should be included in the final executable specifications to ensure that the customers' expectations are met.

Examples should be easy to understand

A common mistake teams make when starting out with Specification by Example is to illustrate requirements using complex and convoluted examples. They focus on capturing realistic examples in precise detail and create huge, confusing tables with dozens of columns and rows. Examples like these make it hard to evaluate consistency and completeness of specifications.

One of the main reasons I prefer examples over abstract statements as requirements is that they allow me to think about functional gaps and inconsistencies. Making things precise makes it easier to spot missing cases. This requires an understanding of the entire set of examples for a particular feature. If the examples aren't easy to understand, we won't be able to evaluate their completeness and consistency. Here are some ideas on how to avoid that problem and still keep the examples precise and realistic.

Avoid the temptation to explore every combinatorial possibility

When teams start illustrating their requirements using examples, testers often misunderstand the purpose of that process and insist on covering every possible combination of arguments. There isn't much point in going through examples that illustrate existing cases; that doesn't improve understanding.

When illustrating using examples, look for examples that move the discussion forward and improve understanding.

I strongly advise against discarding any examples suggested as edge cases without discussion. If someone suggests an edge case example that the others consider to have

been covered already, there might be two possible reasons: Either the person making the suggestion doesn't understand the existing examples, or they have genuinely found something that breaks the existing description that the others don't see. In both cases, it's worth discussing the example to ensure that everyone in the room has the same level of examples suggested as edge casesunderstanding.

Look for implied concepts

When you use too many examples to illustrate a single function or the examples are complex, this often means that the examples should be described at a higher level of abstraction.

Look at the examples and try to identify concepts that are hidden and implied. Make those concepts explicit and define them separately. Restructuring examples like this will make the specifications easier to understand and will lead to better software design.

Looking for missing and implied concepts and making them explicit in system design is one of the core ideas of domain-driven design.[3]

I facilitated a workshop for a team that was rewriting an accounting subsystem and gradually migrating trades from the legacy system to the new product. The workshop was focused on a requirement to migrate Dutch trades to the new system. We started writing examples on a whiteboard and quickly filled all the available space. Looking at the examples, we discovered that we were explaining three things: how to decide which trades are Dutch, how to decide which trades are migrated, and what happens to a trade once it's migrated.

Because we were illustrating all these things at the same time, we had a combinatorial explosion of relevant cases to deal with. When trying to summarize the examples, we identified two implied concepts: a trade location and a migration status. We then broke this requirement into three parts and used a separate, focused set of examples to illustrate each part. We had a specification of how to decide whether a trade is Dutch or not (how to calculate the location of a trade). Another focused set of examples illustrated how the location of a trade affects its migration status. In that set, we used Netherlands only once, without having to go through all the cases that constitute a Dutch trade. The third set of examples illustrated the difference in processing between migrated and non-migrated trades.

[3] Eric Evans, *Domain-Driven Design: Tackling Complexity in the Heart of Software* (Boston, Addison-Wesley Professional, 2003).

Splitting the specification this way allowed the team to significantly improve the design of the system—because three different sets of examples clearly pointed to modular concepts. The next time they had a requirement to migrate a set of trades, they could focus only on changing the definition of a migrated trade. What happens to the trade after it's migrated stays the same. Likewise, the way a trade location is determined doesn't change.

Separating the concepts also facilitated a much more meaningful discussion about trade locations because we were dealing with a small and focused set of examples. We discovered that some people thought the registered location of the company whose stock is being traded determines the location, whereas others thought that only the stock exchange where the company is listed was relevant.

Looking for missing concepts and raising the level of abstraction is no different than what happens in daily communication. Try to give a simple instruction such as "If you come by car, book parking in advance" without using the word *car*; instead, focus on its properties. One way to specify a car is as a transport vehicle with four wheels, four doors, four seats, and a diesel engine. But we also have two-door cars, other types of engines, different numbers of seats, and so on. Listing all those examples would make the instructions ridiculously complicated; instead, we create a higher-level concept to improve communication. How a car is made is irrelevant to parking instructions; what's important is whether the person will arrive by car or not.

Whenever you see too many examples or complicated examples in a specification, try to raise the level of abstraction for those descriptions and then specify the underlying concepts separately.

By illustrating requirements using precise realistic examples and structuring them to be easy to understand, we can capture the essence of required functionality. We also ensure that we've explored the requirements in enough detail for developers and testers to have enough information to start working. These examples can replace abstract requirements in the delivery process and serve as a specification, a target for development, and a verification for acceptance acceptance testing.

Illustrating nonfunctional requirements

Illustrating isolated functional requirements with examples is relatively intuitive, but many teams struggle to do this with functionality that's cross-cutting or difficult to describe with discrete answers. At most of my workshops on Specification by Example, there's usually at least one person who claims that this is possible for "functional" requirements but there's no way that this could work for "nonfunctional" requirements because they aren't that precise.

What are nonfunctional requirements?

Characteristics such as performance, usability, or response times are often called nonfunctional because they aren't related to isolated functionality. I generally disagree with the practice of categorizing requirements as functional or nonfunctional, but that's probably a topic for another book. Many features commonly termed nonfunctional imply functionality. For example, performance requirements might imply a caching function, persistence constraints, and so on. From my experience, what most people think of when they say *nonfunctional* are functional requirements that are cross-cutting (for example, security) or not discrete but measurable on a sliding scale (for example, performance). Dan North points out[†] that requirements listed as nonfunctional usually imply that there's a stakeholder whom the team hasn't yet explicitly identified.

[†] In private communication

So far, I haven't seen a single nonfunctional requirement that couldn't be illustrated using examples. Even usability, perhaps the vaguest and most subjective concept in software development, can be illustrated. Ask your usability expert to show you a website that she likes; that's a good, realistic example. The validation of such examples might not be automatable, but the example is realistic and precise enough to spark a good discussion. Here are some ideas that will help you capture nonfunctional requirements with examples.

Get precise performance requirements

When: Performance is a key feature

Because performance tests often require a separate environment and hardware similar to what's used in production, many performance-critical systems developers can't run any relevant tests on their hardware. This doesn't mean that teams should skip a discussion about performance requirements.

Having the performance criteria clearly specified and illustrated using examples will help build shared understanding and provide the development team with a clear target for implementation.

At RainStor, performance is critical for their data-archiving tools, so they make sure to express the performance requirements in detail. Performance requirements are collected in the form "The system has to import X records within Y minutes on Z CPUs." Developers then either get access to dedicated testing hardware or have the testers run tests for them and provide feedback.

Remember that "faster than the current system" isn't a good performance requirement. Tell people exactly how much faster and in what way.

Use low-fi prototypes for UI

User interface layouts and usability can't be specified easily with examples fitting into truth tables or automated tests. This doesn't mean that we can't discuss examples.

I often create paper prototypes that are glued together from cutouts of user interface elements and website prints. Going through one or two examples is a good way to ensure that we have all the information a customer needs on a screen.

Business users often find it hard to think beyond the user interface, because that's what they work with. This is why boomerangs often happen when a client looks at the software on a screen.

Instead of discussing backend processing, we can sometimes get more concrete information up front by working through a user interface example.

Several teams I interviewed use Balsamiq Mockups,[4] a web/desktop application for low-fi user interface prototyping. I find paper prototypes easier to work with because we can use cutouts and write notes, but a software system works better when we want to share our work.

At RainStor, Adam Knight took this approach even further by creating an interactive prototype to explore vague requirements with clients. He says:

> Rather than a paper prototype we put together some example command line prototype interfaces using shell scripts and then walked these through with the customer, asking them to give us details on how they'd use the new functionality in our system.

This interactive workshop provided functional examples that the development team later used to illustrate requirements. Teams can use this approach to identify scope as well. (See "Don't look only at the lowest level" in chapter 5.)

[4] www.balsamiq.com/products/mockups

Try the QUPER model

When: Sliding scale requirements

When requirements don't lead to discrete, precise results, they're hard to argue about. When was the last time you had a meaningful discussion about why web pages should load in less than two seconds, rather than three seconds or one second? Most of the time, requirements like these are accepted without discussion or understanding.

At the Oresund Developer conference in 2009, Björn Regnell presented QUPER,[5] an interesting model for illustrating requirements that aren't discrete but that work on a sliding scale (for example, startup time or response times). I haven't tried this on a project yet, but because it provides some interesting food for thought, I decided to include it in the book.

QUPER visualizes sliding-scale requirements along the axes of cost, value, and quality. The idea of the model is to estimate cost-benefit breakpoints and barriers on the sliding scale and expose them for discussion.

The QUPER model assumes that such requirements produce benefits on the S curve and that there are three important points on the curve (called breakpoints). *Utility* is the point where a product moves from unusable to usable. For example, the utility point for startup time of a mobile phone is one minute. *Differentiation* describes when the feature starts to develop a competitive advantage that will influence marketing. For example, the differentiation point for mobile phone startup is five seconds. *Saturation* is where the increase in quality becomes overkill. It makes no difference to the user if a phone takes half a second or one second to start, making one second a possible saturation point for mobile phone startup. Regnell argued that going beyond the saturation point means that we're investing resources in the wrong area.

Another assumption of the model is that increases in quality don't lead to linear cost increases. At some point, cost becomes steep. The product might have to be rewritten using a different technology or there will be a significant impact on architecture. These points are called cost barriers in the model.

Defining barriers and breakpoints for sliding scale requirements allows us to have a more meaningful discussion on where the product fits in the market and where we want it to be.

We can use breakpoints and barriers to define relevant targets for different phases of the project and make sliding scale requirements measurable. Regnell suggested setting these as intervals rather than discrete points because this works better with the continuous

[5] See http://oredev.org/videos/supporting-roadmapping-of-quality-requirements and the IEEE Software journal, Mar/Apr 2008.

nature of quality requirements. For example, the target for a feature that just needs to work as well as in competing software should most likely be close to the utility point, definitely not going over the differentiation point. The target for unique selling points of a product should be between the differentiation and saturation points. Visualizing cost barriers on the same curve will help the stakeholders understand how far they can push the targets without having to invest significantly more than ethan expected.

Use a checklist for discussions

When: Cross-cutting concerns

Often, the customers feel safer when they impose a global generic requirement. I've participated in many projects where performance requirements were defined globally; for example, "All web pages will load in less than a second." In most cases, implementing that requirement (and other global requirements like it) is a waste of money. Most often, only the home page and some key functions had to load in less than a second; many other pages could load more slowly. In the QUPER model language, only the loading time of a small number of key pages needs to be close to the differentiation point. Other pages might load in a period of time closer to the utility point.

The problem is that these requirements are defined close to the start of a project, when we still don't know what the product is going to look like.

Rather than taking such requirements at face value, Christian Hassa suggests using these cross-cutting requirements as a checklist for discussions. Hassa says:

> It's easy to specify "The system should respond in 10 milliseconds" globally for the whole system, but you don't necessarily need that level of response time for every feature. What exactly does the system have to do in 10 milliseconds? Does it need to send an email, record the action, or reply? We create acceptance criteria for each feature with this nonfunctional criteria in mind.

A checklist for discussions will ensure that you begin to consider all the important questions when reviewing a story. You can use it to decide which of the cross-cutting concerns apply to a particular story and then focus on illustrating those aspects.

Build a reference example

When: Requirements are impossible to quantify

Because it's subjective and depends on many factors, usability is hard to quantify. But that doesn't mean that it can't be specified by example. In fact, it can only be specified that way.

Usability and similar nonquantifiable features, such as playability and fun, are key for video games. These qualities can't be easily specified with documents that detail traditional requirements. Supermassive Games, a video game studio based in the UK, applies an agile process to game development. The teams at Supermassive use checklists to ensure that different aspects of quality are covered in detail, but that isn't enough to deal with the uncertainty and subjectivity of those features.

Harvey Wheaton, studio director at Supermassive, said that these features have "elusive quality" during his presentation at SPA2010 conference.[6] According to Wheaton, they typically focus on getting one feature finished to the final level of quality early on; then, the team can use that as an example what "done" means:

> We build what we call a "vertical slice" as early on in the process as we can, typically at the end of our pre-production phase. This vertical slice is a small section of the game (e.g., one level, part of a level, the game introduction) and is to final (shippable) quality. This is usually supplemented by a "horizontal slice," i.e., a broad slice of the whole game but blocked out and in low fidelity, to give an idea of the scale and breadth of the game.
>
> You can get a lot of use out of reference or concept art to illustrate the visual look and fidelity of the final product and employ people specifically for this, to produce high quality artwork that shows how the game will look.

Instead of trying to quantify features that have an elusive quality, Supermassive Games builds a reference example against which team members can compare their work.

Building a reference example is an effective way to illustrate nonquantifiable features using examples.

[6] http://gojko.net/2010/05/19/agile-in-a-start-up-games-development-studio

In summary, instead of using the categorization "nonfunctional requirements" to avoid a difficult conversation, teams should ensure they have a shared understanding of what their business users expect out of a system, including cross-cutting concerns. Even if the resulting examples aren't easy to automate later on, having an up front discussion and using examples to make expectations explicit and precise will ensure that the delivery team focuses on building the right product.

Remember

- Using a single set of examples consistently from specification through development to testing ensures that everyone has the same understanding of what needs to be delivered.
- Examples used for illustrating features should be precise, complete, realistic, and easy to understand.
- Realistic examples help spot inconsistencies and functional gaps faster than implementation.
- Once you have an initial set of examples, experiment with data and look for alternative ways to test a feature to complete the specification.
- When examples are complex and there are too many examples or too many factors present, look for missing concepts and try to explain the examples at a higher level of abstraction. Use a set of focused examples to illustrate the new concepts separately.