

---

## Chapter 2

# Finding ways to communicate better

---

In order to bridge the communication gap, we need to work on bringing business people and implementation teams together rather than separating them with formal processes and intermediaries. Ron Jeffries said, during his session on the natural laws of software development at Agile 2008, that the most important information in a requirements document are not the requirements, but the phone number of the person who wrote it. Instead of handing down incomplete abstract requirements, we should focus on facilitating the flow of information and better communication between all team members. Then people can work out for themselves whether the information is complete and correct and ensure that they understand each other.

## Challenging requirements

Since requirements, however clearly expressed, may contain gaps and inconsistencies, how do we fight against this problem before development rather than discovering it later? How do we ensure that requirements, regardless of their form and whether or not they are built up incrementally before every iteration, are complete and correct? Donald Gause and Gerald Weinberg wrote in *Exploring Requirements*[3] that the most effective way of checking requirements is to use test cases very much like those for testing a completed system. They suggested using a black-box testing approach during the requirements phase because the design solution at this point still does not exist, making it the perfect black box. This idea might sound strange at first and it definitely takes a while to grasp. In essence, the idea is to work out how a system would be tested as a way to check

whether the requirements give us enough information to build the system in the first place. We should not dive straight away into how to implement something, but rather think about how the finished system will be used and then double-check the requirements armed with this knowledge. Black-box test scripts are by nature very precise and contain clearly defined steps and values, unlike traditional requirements which are generally a lot more abstract.

On the timeline of a software project, requirements are at the very beginning and black-box tests are traditionally at the end. All the things in between make it hard to see a very subtle relationship between these two concepts. They effectively talk about the same thing: how the system will be used once it is developed. In fact, because tests are very precise, they offer a lot less chance for misunderstanding than abstract requirements. This makes tests theoretically an option for replacing requirements altogether. Robert C. Martin and Grigori Melnik even consider them the same thing in their *equivalence hypothesis*[8]:

*As formality increases, tests and requirements become indistinguishable. At the limit, tests and requirements are equivalent.*

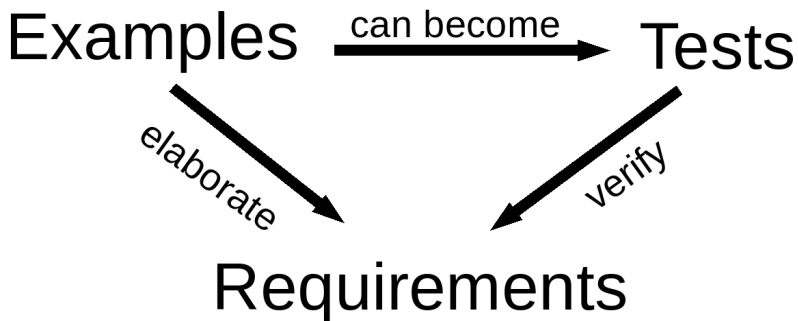
## We really communicate with examples

In order to identify requirements, business analysts often work through a number of realistic examples with the customers, such as existing report forms or work processes. These examples are then translated to abstract requirements (equivalent to the the first step of the telephone game). An interesting thing about examples is that they pop up several times later in the process as well. Abstract requirements and specifications leave a lot of space for ambiguity and misunderstanding. In order to verify or reject ideas about the requirements, developers often resort to examples and try to put things in a more concrete perspective when talking about edge cases

to business experts or customers. Concrete realistic examples give us a much better way to explain how things really work than requirements. Examples are simply a very effective communication technique and we use them all the time; this comes so naturally that we are often not aware of it. Test scripts that are produced to verify the system are also examples of how the system behaves. They capture a very concrete workflow, with clearly defined inputs and expected outputs.

The participants of the second Agile Alliance Functional Testing Tools workshop, held in August 2008 in Toronto, came up with a diagram<sup>1</sup> explaining the relationship between examples, requirements and tests shown in Figure 2.1. Examples, requirements and tests are essentially tied together in a loop.

**Figure 2.1. Relationship between examples, tests and requirements**



This close link between requirements, tests and examples signals that they all effectively deal with related concepts. The problem is that every time examples show up on the timeline of a software project, people have to re-invent them. Customers and business analysts use one set of examples to discuss the system requirements. Developers come up with their own examples to talk to business analysts. Testers come up with their own examples to write test scripts. Because of the effects illustrated by the telephone game, these examples might not

---

<sup>1</sup>Many thanks to Jennitta Andrea for pointing this out to me

describe the same things (and they often do not). Examples that developers invent are based on their understanding. Test scripts are derived from what testers think about the system.

Going back to the equivalence hypothesis, tests and requirements can be the same. Requirements are often driven from examples, and examples also end up as tests. With enough examples, we can build a full description of the future system. We can then discuss these examples to make sure that everyone understands them correctly. By formalising examples we can get rigorous requirements for the system and a good set of tests. If we use the same examples throughout the project, from discussions with customers and domain experts to testing, then developers or testers do not have to come up with their own examples in isolation. By consistently using the same set of examples we can eliminate the effects of the telephone game.

## Working together, we find better solutions

Here is an example that shows how communication across project roles can lead to a better solution. During the Crevasse of Doom<sup>2</sup> presentation at the QCon conference in London in March 2007, Dan North talked about a project in the early days of Java, when printing documents was extremely hard to implement properly. The developers told the business people that all the requirements could be easily implemented, apart from those related to printing because printing APIs were not mature enough. The business people refused to give in because “printing was embedded in the core of their business”. While discussing this further and talking about how exactly printing was embedded in the core of the business, the developers discovered that the users entered data in a screen, printed out the results of calculations, opened a different screen and entered these results, processed and printed results again, entered them in the third screen and so on. After seeing this, the developers suggested automatically

---

<sup>2</sup><http://www.infoq.com/presentations/Fowler-North-Crevasse-of-Doom>

transferring the information between screens, invoking the amazed response “You can do that?”. Although this example is rather simplistic, it illustrates a very important fact: neither group could solve the problem on their own, but together they found a great solution. The business users were so used to working around a technical difficulty that they really thought of printing as something embedded in the process. The developers did not really understand what was going on or why printing was needed.

If we help each other understand the goal better, then we will come up with better solutions. This is why building a shared understanding of the goal is one of the key practices in software development.

## Communicating intent

The communication of goals was at the core of Prussian military tactics as a response to the dangers posed by Napoleon's invincible army. The Prussian leaders figured out that they did not have a single person capable of defeating Napoleon's genius, so they focused on allowing the individual commanders and their troops to act collectively to better effect. They made sure they told the commanders why something needed to be done and they put this information into the perspective of overall goals, rather than just passing down a list of imperative commands. The resulting military doctrine was called *Auftragstaktik*, or mission-type tactics, and it was key to the great successes of Prussian and later German armies. Today it survives as Mission Command in the US Army.

A key lesson to take from this and apply in software development is that understanding business reasons behind technical requests is crucial for building a shared understanding of the domain. Examples in the previous section and the section *Requirements are often already a solution* on page 18 demonstrate this. *Original intent is one of the most important things that a customer or a business analyst should pass on to the developers and testers.* At the same time, in my experience, passing on this information is one of the most underestimated and neglected practices in software development. This is just as much

a problem in other areas. Going back to the example with US Army orders from the section *Imperative requirements are very easy to misunderstand* on page 8, effectively communicating the intent was one of the biggest problems that researchers identified. Gary Klein cites research by William Crain,<sup>3</sup> who found that only 19% of commanders' intent statements had anything to say about the purpose of the mission and that "communication of intent was mediocre". When domain experts reviewed the statements, the average score for effectiveness was below the average, "closer to very ineffective".

I was recently involved in a project where the clients said that they wanted real-time reporting on transactions in a highly distributed system. The system was being developed for batch reporting and introducing real-time reporting required a huge redesign, possibly even a major change to the infrastructure. Instead of challenging this requirement, the CTO of the company started to design a solution on the spot, immersing himself in a tough technical problem which, for him, was a pleasure to play with. I asked what the clients wanted to achieve with real-time reports and we came to the conclusion that they wanted to present up-to-date information to their partners during the working day, not just overnight. A delay of one hour was perfectly acceptable for this, without the need to redesign the system from scratch or invest a lot of money in real-time infrastructure, especially since real-time reporting on live transactional data would have seriously impacted the performance of the transaction processing service. In this case, the issue was with the meaning of word 'real-time'. For the customers, it meant 'updated several times per day' but for programmers it meant 'millisecond correctness'.

The moral of this story is that requirements should not really be taken at face value, especially if they come in the form of a solution that does not explain what the intent is (see the section *Requirements are often already a solution* on page 18). Teams often consider requirements as something carved in stone that has to match exactly what the customers demand. In fact, customers will often accept a different

---

<sup>3</sup> See page 225 of [4] and <http://handle.dtic.mil/100.2/ADA225436>

solution if it solves the problem, especially if it is a lot cheaper and easier to implement.

Understanding why something is required, instead of just blindly following instructions, is definitely one of the key factors for improving the chances of success in a software project. *This is why it is crucial to communicate the reasons behind technical requests and business decisions.* Don't be afraid to ask 'why' if you do not understand a requirement or if you find it strange.

## Agile acceptance testing in a nutshell

We can bridge the communication gap by communicating intent, getting different roles involved in nailing down the requirements and exploiting the relationships between tests, examples and requirements. We can use realistic examples consistently throughout the project to avoid the translation and minimise the effect of the telephone game, saving information from falling through the communication gaps. In order to ensure that the initial set of examples is good enough for development and testing, programmers and testers have to get involved in defining them. The discussion about these examples builds up a shared understanding of the domain, flushing out small communication problems straight from the start.

This is essentially what agile acceptance testing is about. Over the years, it has evolved into a relatively simple and elegant practice. (Please do not confuse simple and elegant with easy. Implementing agile acceptance testing can be a real organisational challenge and may require a lot of effort. But it will pay off in a big way at the end.) It revolves around the following five ideas:

1. Use real-world examples to build a shared understanding of the domain.
2. Select a set of these examples to be a specification and an acceptance test suite.

3. Automate verification of the acceptance tests.
4. Focus the software development effort on the acceptance tests.
5. Use the set of acceptance tests to facilitate discussion about future change requests, effectively going back to step 1 for a new cycle of development.

## Use real-world examples to build a shared understanding of the domain

Instead of abstract requirements that can easily be misinterpreted, we should talk about what the system does or should do in real-world situations. Abstract definitions can obscure our view and leave many ambiguities. Real-world examples do not suffer from this. Discussing realistic situations helps flush out assumed business rules and points us to the true business rules. Ideally, we want to identify representative cases that show all the differences in possibilities and then discuss these cases and how the system should behave. Writing the examples down on a whiteboard, wall, word document or a piece of paper will help discover gaps and identify additional scenarios and examples that we have to discuss. Working with real-world examples helps us communicate better because people will be able to relate to them more easily. It is also easier to spot inconsistencies between realistic examples. Developers, business people and testers all need to participate in the discussion about examples. Developers learn about the domain and get a solid foundation for implementation. Testers obtain the knowledge they need firsthand and they can influence the development by suggesting important cases for discussion. I explain this step in greater detail in Chapter 3 and Chapter 4.

## Select a set of these examples to be a specification and an acceptance test suite

Once we have identified enough examples for everyone to be comfortable with starting the implementation, we can actually use the examples as the acceptance criteria for the current iteration of

development. A selected set of examples can become the target for development – effectively the solution specification. Once the system does exactly what we discussed in all of the cases, it is ready to be shipped. Verifying all the cases becomes the final check-point before a release. This step is explained in greater detail in Chapter 5.

## Automate verification of the acceptance tests

If we are moving away from the target, I'd like to know this sooner rather than later. So we want to be able to check the examples often, to make sure that the code is going towards the target. The biggest expense in software development today is the time of the implementation team members (again, I'm using this as a name for all participants in the development process, not just developers), so verifying the examples must not take a lot of time. In addition, automated tests provide quick feedback and make the rest of our work more efficient. This is why we need to automate the verifications. When all the examples are automated, we can check whether the code does what we expected with a touch of a button. This step is explained in greater detail in Chapter 5.

## Focus the software development effort on the acceptance tests

At the end of the project all the examples should work. Because we ideally have a complete set of examples, if all the examples work there is nothing else that should be developed. So we can actually focus the development on satisfying the examples. After the examples are automated, we can verify the code quickly and often. Develop a bit of code to satisfy an example, run the verification and check whether we are really doing it correctly, then move on to the next example. This helps to avoid just-in-case code, which is what happens when developers have to think about all the edge cases to satisfy an abstract requirement. There are no abstract requirements, and we have

examples to describe all the edge cases. When all the examples are implemented so that the system works as they describe, the job is done. To developers this might sound similar to unit testing, because it essentially tries to do the same thing on a higher level (but acceptance tests are not a replacement for unit tests). I talk about this step in more detail in Chapter 6.

## Use the acceptance tests to facilitate future change

Once the development is done, change requests will start coming in. We can use the set of acceptance tests for previous phases of development as a relevant authoritative document on the system. We use them to discuss change requests and quickly discover conflicting rules and changes. This is explained in more detail in Chapter 7.

## And repeat

These steps are continuously repeated throughout the project to clarify, specify, implement and verify small parts of the project iteratively and incrementally. In Chapter 8 I explain how this fits into the overall development process in more detail.

## So what does testing have to do with this?

On the first page of the introduction I said that agile acceptance testing is a practice that improves communication in the team. I also talked about focusing development and producing better specifications. This often raises the question of what all this has to do with testing. In fact, the name agile acceptance testing is a bit misleading and one of the main causes for so many misconceptions about this practice. The name comes from the fact that this practice evolved from test-driven development (TDD). Don't confuse it with customer

(or user) acceptance testing, which is a practice that was established a long time before agile acceptance testing. I explain the differences between these two later.

In test-driven development, developers write tests to specify what a unit of code should do, then implement the code unit to satisfy the requirement. These tests are called unit tests, as they focus on small code units. The benefits of this approach are a better focus on what the system should do and having a clearly defined target for development. The team can collectively own the code and verify it, so this practice facilitates better team work. Unit tests also hold the system together during change, allowing it to be more flexible. If a change unintentionally breaks some existing functionality, the relevant unit test will fail and alert developers about a problem.

The practice of unit testing was so useful that it was only logical to ask whether the same could be applied to business rules and drive whole phases of projects rather than just code units. Unit tests focus on code so they are completely in the domain of software developers. Business rules, on the other hand, should not be defined by developers. They have to be defined by domain experts and customers. But business domain experts rarely understand programming languages, so using the same tools to drive implementation of business rules and code units typically fails. Developers could write business rule tests with unit test tools, but these tests would only reflect what developers understand and they would still be affected by all the communication problems presented in Chapter 1. In order to get the best specifications, customers and implementation teams have to work together. There is simply no way for a business person to verify that a developer's test describes the end goal correctly with unit test tools. So, in theory, the ideas behind unit testing could be applied to business rules as well, but in practice the tests are impossible to communicate.

To solve the problem, better ways for specifying and automating tests for business rules were needed that could also be used to communicate with business people. Such tools would have to focus on capturing the customers' view of what the system should do when it is finished, effectively the functional acceptance criteria for the project. Then we

could apply unit testing ideas by developing code to satisfy tests and running these tests to verify that the code is on target, then repeating the process until all tests go green. This is where the name agile acceptance testing comes from. This practice started by expanding the unit testing ideas to business rules, effectively specifying the acceptance criteria in a form that could be executed as tests on the code.

Going back to the conclusion of the previous chapter, the problem is essentially a communication issue, not a technical one. The biggest obstacle for any such effort is communication, so easy communication and collaboration with business people have to be part of any viable solution. Out of the effort to solve the problem have come tools like FIT, FitNesse and so on. The ideas applied in these tools solved the communication problem so nicely that the practice of agile acceptance testing evolved into a great way to build a shared understanding of the domain and enable all project participants to speak the same language. Testing, in any possible meaning of this word, becomes relatively unimportant because the greatest benefit is improved communication.

Today, the name of this practice itself has become a major obstacle to its adoption. The chief business analyst of a company I recently worked with just rejected getting involved in the practice, with the explanation “I do not write tests”. The word testing unfortunately bears a negative connotation in the software development world. In all the companies I worked for, testers were among the least well-paid employees, right down there with the support engineers. Starting a discussion on testing somehow seems to give business people the green light to tune out. It is like a signal that the interesting part of the meeting is over and that they can start playing Sudoku or thinking about more important things. After all, testing is not something that they do.

The practice of agile acceptance testing, especially as a way to improve communication between team members and build a shared understanding of the domain, relies on the participation of business people. They are the ones that need to pass their domain knowledge on to

programmers and testers. They are also the ones that need to make decisions about edge cases and answer tough questions about business rules. So they very much have to 'do tests'.

To add a further complication, there is also user acceptance testing, which sounds very similar to agile acceptance testing. User acceptance testing (UAT) is a phase in a software project where clients sign off a deliverable and accept it as complete. It can involve testing by end-users, verification from stakeholders that they are happy with the product and a whole range of other testing activities. Although you can verify the software during UAT using a successful run through an acceptance testing suite as one of the criteria, agile acceptance testing has very little to do with user acceptance testing. In fact, they are on completely opposite sides of a software project. User acceptance testing happens after development and it is often the final confirmation before money changes hands. It is performed by the clients or third-party agents. Agile acceptance testing happens before and during development and it is performed by the implementation team.

Some people call this practice acceptance test-driven development to signify that acceptance tests are used to drive the development, not just to verify the deliveries at the end. Others try to reduce the confusion between user acceptance testing and agile acceptance testing by avoiding the use of the word 'acceptance' and calling the practice functional test-driven development. This also signals the difference between code-oriented unit tests which are traditionally linked with TDD and functional (acceptance) tests. Three or four years ago, the name storytest-driven development was used to describe this practice, emphasising the fact that acceptance tests are related to user stories and not code, but it did not really catch on. Another variant is test-driven requirements, explaining that acceptance tests actually deal with requirements more than with development. Brian Marick calls these tests business-facing to emphasise that business people should be concerned with them.<sup>4</sup>

---

<sup>4</sup><http://www.exampler.com/old-blog/2003/08/21/>

## Better names

Dan North suggests using the word ‘behaviour’ instead of ‘test’,<sup>5</sup> as a way to clear up a lot of misunderstandings. Instead of test-driven development, he talks about *behaviour-driven development* to avoid the negative connotation of testing. This trick has solved the problem of keeping business people awake quite a few times for me as well. Behaviour-driven development (BDD) is just a variant of agile acceptance testing, in my opinion. Some people will disagree, pointing out the differences in tools and format of test scripts. For me, the underlying principles are the same and BDD tools are just another way to automate tests. BDD also promotes a specific approach to implementation,<sup>6</sup> but this is not really important for the topic of this book. Again, I consider tools and tests to be of less importance than communication and building a shared understanding.

A name that has become more popular recently is *example-driven development*, with tests being called *specification by example*. This reflects the fact that we are using concrete real-world examples to produce the specifications instead of abstract requirements. I also like *executable specifications* as a name instead of acceptance tests, because it truly describes the nature of what we are building. Agile acceptance tests are specifications for development in a form that can be verified by executing them directly against the code.

People have tried to rename agile acceptance testing several times, but this name has somehow stuck. Most still use this name and this is why I decided to keep it here. I mention all the alternative names here because they are interesting attempts to remove the word ‘test’ from the vocabulary to reduce the ambiguity and misunderstanding that come from it. If your business people don't want to participate because testing is beneath them, try to present the same thing but with a different name.

---

<sup>5</sup><http://dannorth.net/introducing-bdd>

<sup>6</sup> See <http://behaviour-driven.org/> for more information

In any case, I want to point out that there are a lot of different names and ideas emerging at the moment, but they are all effectively different versions of the same underlying practice. This book is about the underlying values, practices and principles that all these names and ideas share.



### *Stuff to remember*

- Realistic examples are a great way to communicate, and we use them often without even thinking about it.
- Requirements, tests and examples all talk about the same thing – how a system will behave once it is delivered.
- We can use examples consistently throughout the project to avoid the effects of the telephone game.
- Building a shared understanding of the problem is one of the key practices in software development.
- Business intent is one of the most important things that a customer or a business analyst should pass on to the developers and testers.
- Cross-functional teams create much better specifications and requirements than business people in isolation.
- Agile acceptance testing uses these ideas to solve communication problems on software projects.
- The name agile acceptance testing is misleading but has been generally adopted.
- Agile acceptance testing is very different from user acceptance testing, and in general it is not about testing at all. It is about improving communication and building a shared understanding of the domain.
- Implementing agile acceptance testing can be a real organisational challenge.

Agile acceptance testing in a nutshell revolves around these five principles:

1. Use real-world examples to build a shared understanding of the domain.
2. Select a set of these examples to be a specification and an acceptance test suite.
3. Automate the verification of acceptance tests.
4. Focus the software development effort on the acceptance tests.
5. Use the set of acceptance tests to facilitate discussion about future change requests, effectively going back to step 1 for a new cycle of development.