university of groningen / Faculty of Science and Engineering / computing science

Problem Analysis & Software Design

Jan Salvador van der Ven, Rein Smedinga

2021 – 2022

Problem Analysis & Software Design

This reader was composed by:

Jan Salvador van der Ven
With help of Trias Informatica

Special thanks

The styling of this document is inspired by the reader style defined in L-Space.

Revision number: 1.106
Revision date: 27th October 2021

Disclaimer

The content of this syllabus will be subject to changes.

Table of Contents

Section 0

Introduction to the course

This is the syllabus for the course problem analysis and software design. The details of this course can be found on Nestor.

For feedback on this syllabus, please contact the teacher of this course at: mail@jansalvador.nl. All kind of feedback is welcome, from textual enhancements, layout issues to suggestions on the content.

Section 1

Literature

The literature in this chapter is composed from several books and other sources. This chapter described the information about the literature's origin, listed in order of occurrence.

Robertson, S., & Robertson, J. (1999). Mastering the requirements process. Harlow: Addison-Wesley.

Leffingwell, D. (2011). Agile software requirements: Lean requirements practices for teams, programs, and the enterprise.

1. Some Fundamental Truths

in which we consider the essential contribution of requirements

Truth 1

Requirements are not really about requirements.

Requirements are what the software product, or hardware product, or service, or whatever you intend to build, is meant to do and to be. Requirements exist whether you discover them or not, and whether you write them down or not. Obviously, your product will never be right unless it conforms to the requirements, so in this way you can think of the requirements as some kind of natural law, and it is up to you to discover them.

That said, the requirements activity is not principally about writing a requirements document. Instead, it focuses on understanding a business problem and providing a solution for it. Software is there to solve some kind of problem, as are hardware and services. The real art of requirements discovery is discovering the real problem. Once you do that, you have the basis for identifying and choosing between alternative solutions. In essence, then, requirements are not about the written requirements as such, but rather an uncovering of the problem to be solved.

Incidentally, when we say “business,” “business problem,” or “work” we mean whatever activity you are concerned with—be it commercial, scientific, embedded, government, military, or, indeed, any other kind of activity or service or consumer product.

Throughout this book we have used “he” to refer to both genders. The authors (one male and one female) find the use of “he or she” disruptive and awkward.

Also incidentally, when we say “he” in this book—usually referring to the business analyst—we mean “he or she.” We find it too clumsy to keep saying “he or she” or “he/she.” Believe us, requirements work belongs equally to both genders.

Truth 2

If we must build software, then it must be optimally valuable for its owner.

Note that we are concerned with the owner of the end result, and only indirectly the user. This focus seems to run contrary to the usual priorities, so we had best explain it.

The owner is the person or organization that pays for the software (or hardware or any other product you might be building). Either the owner pays for the development of the software or he buys the software from someone else. The owner also pays for the disruption to his business that happens when the software is deployed. On the other side of the ledger, the owner gets a benefit from the software. To describe that relationship very simply, the owner is buying a benefit.

We could say that another way—the owner will not pay unless the product provides a benefit. This benefit usually comes in the shape of providing some capability that was not previously available, or changing some business process to be faster or cheaper or more convenient. Naturally this benefit must provide a value to the owner that exceeds the cost of developing the product (see Figure 1.1).

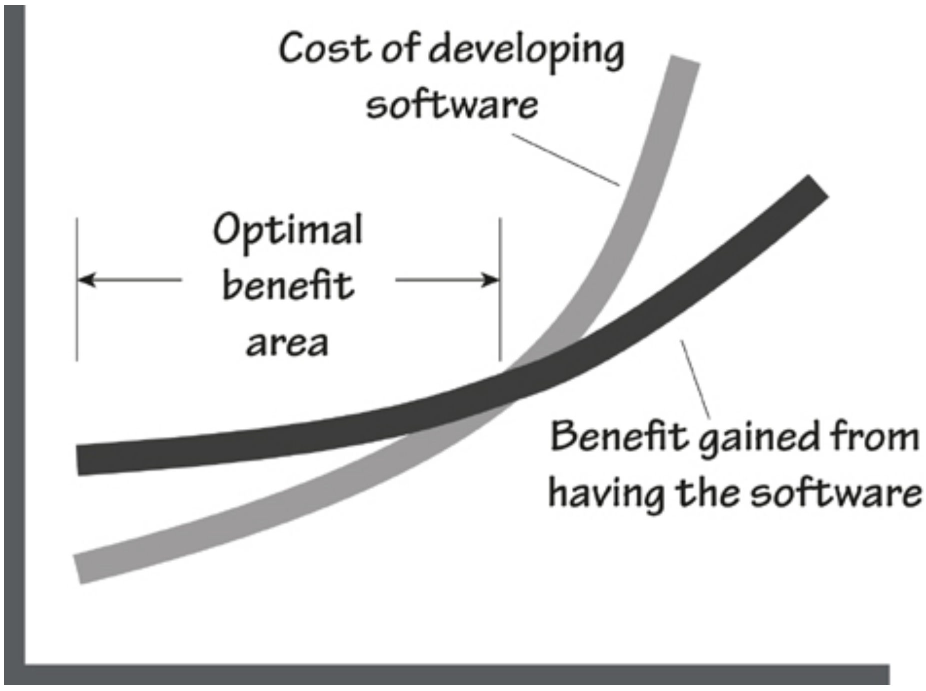


Figure 1.1. As the software becomes more capable and the cost of construction increases, so does the benefit that the software brings. At some point, however, the cost of construction starts to outstrip the benefit and the project is no longer beneficial.

To be *optimally valuable,* the product must provide a benefit that is in proportion to the cost of the product. In some cases, the product can have a very high cost if the value to the owner is great enough. For instance, airlines are willing to pay significant sums for simulators that ensure their pilots are suitably qualified and skilled; lives will be lost if they aren't. An airline might also pay a lot for an automated check-in system when it will make significant inroads into the cost of getting passengers onto planes. The same airline would pay far less for a canteen staff roster system because, let's face it—that kind of task can be done manually and having a few wrong people in the canteen is annoying but hardly life-threatening.

The role of the requirements discoverer—call him a "business analyst," "requirements engineer," "product owner," "systems analyst," or any other title—is to determine what the owner values. In some cases, providing a small system that solves a small problem provides sufficient benefit for the owner to consider it valuable. In other cases (perhaps many others), extending the system's capabilities will provide a much greater value, and this can be achieved for a small additional cost. It all depends on what the owner values.

This, then, is optimal value—understanding the owner's problem well enough to deliver a solution that provides the best payback at the best price.

Truth 3

If your software does not have to satisfy a need, then you can build anything. However, if it is meant to satisfy a need, then you have to know what that need is to build the right software.

It is worthwhile considering that the most useful products are those for which the developers correctly understood what the product was intended to accomplish for its users, and in what manner it was to accomplish that purpose. To understand these things, you must understand the work of the owner's business and determine how that work should be carried out in the future.

Once these points are understood and agreed to, then the business analysts negotiate with the owner about which product will best improve the work. The business analysts produce requirements that describe the functionality of the product—what it will do; and the quality attributes of the product—how well it will do it.

Without knowing these requirements, there is little chance that any product emerging from the development project will be of much value. Apart from a few fortuitous accidents, no product has ever succeeded without prior understanding of its requirements.

It does not matter which kind of work the owner wishes to do, be it scientific, commercial, e-commerce, or social networking. Nor does it matter which programming language or development tools are used to construct the product. The development life cycle—whether agile, prototyping, spiral, the Rational Unified Process, or any other method—is irrelevant to the need for understanding the requirements.

This truth always emerges: You must come to the correct understanding of the requirements, and have your client agree with them, or your product or your project will be seriously deficient.

"On two occasions I have been asked, 'Pray, Mr. Babbage, if you put into the machine wrong figures, will the right answers come out?' I am not able rightly to apprehend the kind of confusion of ideas that could provoke such a question."

—Charles Babbage

Sadly, the requirements are not always correctly understood. Authors Steve McConnell and Jerry Weinberg provide statistics showing that as many as 60 percent of errors originate from within the requirements activity. Software developers have the opportunity to (almost) eliminate these errors. Yet many choose—or their managers choose—to (almost) eliminate the requirements discovery and rush headlong into constructing the (inevitably) wrong product. As a result, they pay many times the price for their product than they would have if the requirements discovery had been done correctly in the first place. Poor quality is passed on in the development life cycle; it is as simple as that.

Truth 4

There is an important difference between building a piece of software and solving a business problem. The former does not necessarily accomplish the latter.

Many software development projects concentrate solely on the software. This might seem reasonable—after all, most software projects manage to produce some software. However, concentrating almost exclusively on the software is a little like trying to build the Parthenon by concentrating on stones. The software, if it is to be valuable to the owner, must solve the owner's business problem.

We build an awful lot of software. Tens (if not hundreds) of millions of lines of code are produced each year. Much of this output contains errors, and most of those are errors of requirements. As a consequence, an awful lot of the world's software simply does not solve the correct problem.

Some development processes are based on the idea of delivering some functionality to its intended users, and inviting them to say whether it solves their problem. If it does not, the software is reworked, and then once again presented for approval. The problem here is that we never know whether the users approve the last delivery because they are satisfied with it or because they are exhausted by the process.

More importantly, it is very difficult for an individual user to understand the broader ramifications of deploying a piece of software. Typically software users do not know enough about the wider business to decide whether this incarnation of software will cause problems in some other part of the business.

And at the risk of repeating ourselves, we cannot stress enough that software is there to solve a business problem. Clearly, then, any development effort must start with the problem, and not with a perceived solution.

Truth 5

The requirements do not have to be written, but they have to become known to the builders.

It often seems that the aim of a requirements project is to produce as large a specification as possible. It doesn't seem to matter that very few readers can understand much of it, and that even fewer have the patience to read it. It appears that the requirements writers believe that their work will be appreciated in direct proportion to the thickness of the specification.

Once produced, this considerable document is then thrown over the wall—or, should we say, forklifted over the wall—to the developers, who are expected to rejoice at the sheer volume of the specification. After all, the more pages it contains, the more chance it has not missed anything—or so the theory goes. Naturally enough, the developers are almost always underwhelmed by this document and either to ignore it or willfully comply with it. Either way, the end result is usually unsatisfactory.

Despite this bizarre behavior, there remains a need for requirements, and for those requirements to be communicated to the development team (see Figure 1.2).

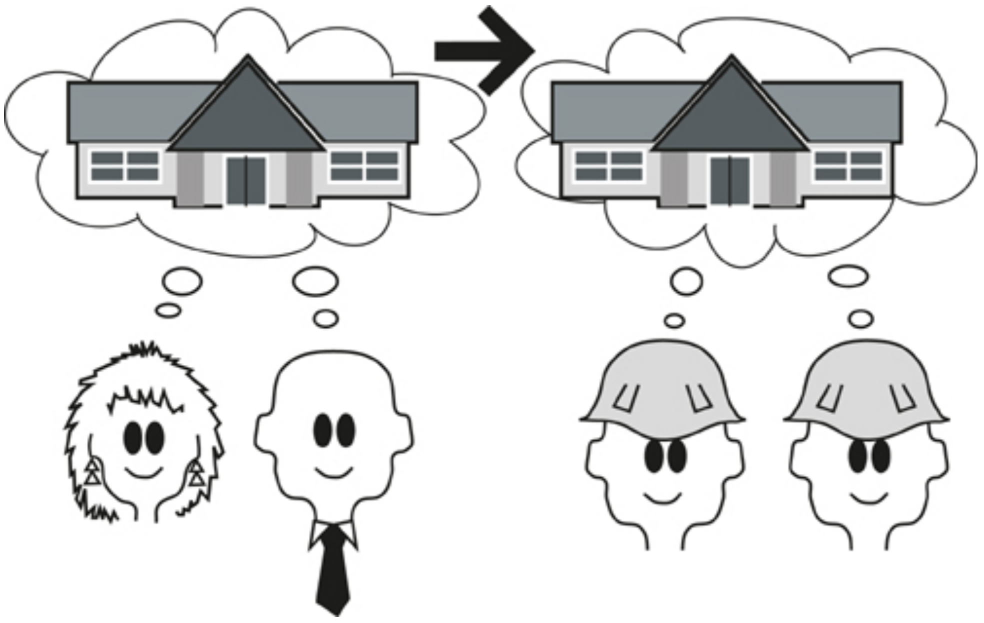


Figure 1.2. Naturally, there is a need to communicate the requirements to the builders of the product.

Whether the requirements are written or not is beside the point. In some cases it is more effective to verbally communicate requirements; in other cases there is an inescapable need for a permanent record of the requirements.

Despite the efficacy of verbal requirements, we feel that it is not feasible to communicate *all*requirements this way. In many cases the act of writing a requirement helps both the business analyst and the stakeholder to completely understand it. As well as improving the understanding, a correctly written requirement provides trace documentation. The rationale of a requirement, or the justification on a story card, documents the team's decisions. It also provides the testers and the developers with a clear indication of the importance of the requirement, which in turn suggests how much effort to expend on it. Additionally, the cost of future maintenance is reduced when the maintainers know why a requirement exists.

Requirements are not meant to place an extra burden on your project, so nothing should be written unless there is a clear need for it. Nevertheless, when the need exists, then the effort involved in writing a requirement is paid back several times over by the precision of the requirement and the reduction in the maintenance effort that is yet to come.

Truth 6

Your customer won't always give you the right answer. Sometimes it is impossible for the customer to know what is right, and sometimes he just doesn't know what he needs.

The requirements activity is traditionally seen as somewhat akin to the task of a stenographer. That is, the business analyst listens carefully to the stakeholders, records precisely whatever it is they say, and translates their requests into requirements for the product.

The flaw in this approach is that it does not take into account the difficulty stakeholders have when they are trying to describe what they need. It is no simple task to envisage a product that will solve a problem, particularly when the problem is not always completely understood. Given the complexity and scale of today's businesses, it is very difficult, indeed, for individuals to understand all appropriate parts of the business.

We also have the problem of *incremental improvements*. It is far too common that when asked about a new system, stakeholders describe their existing system, and add on a few improvements. This incremental approach generally precludes any serious innovation, and it often results in mediocre products that fail to live up to expectations.

The business analyst has to perform a juggling act. In some cases he must record the customer's request; in some cases he must persuade the customer that what is being asked for is not what is needed; in other cases he has to derive the requirements from the customer's solution; and in some cases he must come up with an innovation that is not what anyone asked for, but results in a better solution. In all cases, he should think that any stakeholder could be Pinocchio (Figure 1.3) and not believe everything he is told.

Figure 1.3. Sometimes, like Pinocchio, your customer does not tell you the whole truth.

Truth 7

Requirements do not come about by chance. There needs to be some kind of orderly process for developing them.

Any important endeavor needs an orderly process. Random applications of steel and concrete do not produce buildings; there is a defined process for designing and erecting such structures. Similarly, there is a defined and systematic process for making movies. Your motorcar was designed and built using orderly processes, and your last airline flight was the result of a set of orderly processes that were followed more or less verbatim. Even artistic endeavors, such as novels and paintings, have an orderly process that the artist follows.

These processes are not lockstep procedures where one mindlessly follows every instruction without question, in the prescribed sequence, and without variation.

Instead, orderly processes comprise a set of tasks that achieve the intended result, but leave the order, emphasis, and degree of application to the person or team using the process.

Most importantly, the people who are active in the process must be able to see why different tasks within the process are important, and which tasks carry the most significance for their project.

Truth 8

You can be as iterative as you want, but you still need to understand what the business needs.

Since the previous edition of this book was published, iterative development methods have become much more popular. This is certainly a worthwhile advance, but like many advances these techniques are sometimes over-hyped. For example, we have heard people say (and some commit to print) that iterative delivery makes requirements redundant.

Cooler heads have realized that any development technique has a need to discover the requirements as a prerequisite to serious development. In turn, the cooler heads have absorbed requirements processes into their development life cycles. Instead of attempting to do away with requirements, intelligent methods simply approach the requirements need from a different direction.

The real concern—and this applies to any kind of development technique—is to discover what is needed without producing unnecessary, premature, and wasteful reams of documentation.

No matter how you develop your software, the need to understand the customer's business problem, and what the product has to do to solve this problem (in other words, its requirements), remains.

Truth 9

There is no silver bullet. All our methods and tools will not compensate for poor thought and poor workmanship.

While we have a need for an orderly process, it should not be seen as a substitute for thinking. Processes help, but they help the smart people much more than they help people who are not prepared to think. This is particularly true of requirements processes where the business analyst is required to juggle several versions of the requirements, and at the same time imagine what will make the best future software product.

The requirements activity is not exactly easy; it takes thought and perception on the part of the business analyst if it is to succeed. Several automated tools are available to help with this endeavor, but they must be seen as aids and not as substitutes for good requirements practices. No amount of blindly following a prescribed practice will produce the same result as a skilled business analyst using his most important tools—the brain, the eyes, and the ears.

"[E]ven perfect program verification can only establish that a program meets its specification. The hardest part of the software task is arriving at a complete and consistent specification, and much of the essence of building a program is in fact the debugging of the specification."

—Fred Brooks, *No Silver Bullet: Essence and Accidents of Software Engineering*

Truth 10

Requirements, if they are to be implemented successfully, must be measurable and testable.

At its heart, a functional requirement is something that your product must do to support its owner's business. A non-functional requirement is the quantification of how well it must carry out its functionality for it to be successful within the owner's environment.

To build a product that exactly meets these criteria, you must be precise when writing requirements. At the same time, you must take into account the fact that requirements come from humans, and humans are not always, and sometimes never, precise. To achieve the necessary level of precision, you have to somehow measure a requirement. If you can measure the requirement using numbers instead of words, you can make the requirement testable.

For example, if you have a requirement that your product "shall be attractive to new users," then you can set a measurement that a first-time user can successfully set up an account within 2 minutes, with less than 5 seconds' hesitation for any item of data for which the user is expected to know—such as his name, e-mail address, and similar items. (Hesitation time is a measure of how intuitive the product is, which is part of its attractiveness to the user.) Naturally, when you measure the requirement this way, your testers can determine whether the product (or in some cases a prototype of the product) meets its need.

It is also safe to say that if you cannot find a measurement for a requirement, then it is not a requirement, but merely an idle thought.

Truth 11

You, the business analyst, will change the way the user thinks about his problem, either now or later.

When you come to understand the requirements, especially when they come from different stakeholders, you start to build abstractions and establish vocabulary. When you present models of the business processes, when you work with the stakeholders to find the essence of the work, when you have clear and measurable requirements, and when you reflect all this truth back to the stakeholders, it will change (for the better) their thinking about their business problem.

Once people have a better understanding of the real meaning of their requirements, they are likely to see ways of improving them. Part of your job is to help people, as early as possible, to understand and question their requirements so that they can help you to discover what they really need.

What Are These Requirements Anyway?

After all that truth, what are these requirements that we keep talking about? Simply put, a requirement is something the product must do to support its owner's business, or a quality it must have to make it acceptable and attractive to the owner. A requirement exists either because the type of product demands certain functions and qualities, or because the client justifiably asks for that requirement to be part of the delivered product.

Functional Requirements

A functional requirement describes an action that the product must take if it is to be useful to its operator—they arise from the work that your stakeholders need to do. Almost any action (calculate, inspect, publish, or most other active verbs) can be a functional requirement.

Functional requirements are things the product must do.

The product shall produce a schedule of all roads upon which ice is predicted to form within the given time parameter.

This requirement is one of the things that the product must do if it is to be useful within the context of its owner's business. You can deduce that owner in this case is an organization that is responsible for maintaining roads safely, and that does so by dispatching trucks to spread de-icing material on roads where ice is about to form.

Non-functional Requirements

Non-functional requirements are properties, or qualities, that the product must have if it is to be acceptable to its owner and operator. In some cases, non-functional requirements—these specify such properties as performance, look and feel, usability, security, and legal attributes—are critical to the product's success, as in the following case:

Non-functional requirements are qualities the product must have.

The product shall be able to determine "friend or foe" in less than 0.25 second.

Sometimes they are requirements because they enhance the product or make people want to buy it:

The product shall provide a pleasing user experience.

Sometimes they make the product usable:

The product shall be able to be used by travelers in the arrivals hall who do not speak the home language.

Non-functional requirements might at first seem vague or incomplete. Later in this book we will look at how to give them a fit criterion to make them measurable and thus testable.

Constraints

Constraints are global requirements. They can be limitations on the project itself or restrictions on the eventual design of the product. For example, this is a project constraint:

Constraints are global issues that shape the requirements.

The product must be available at the beginning of the new tax year.

The client for the product is saying that the product is of no use if it is not available to be used by the client's customers in the new tax year. The effect is that the requirements analysts must constrain the requirements to those that can deliver the optimal benefit within the deadline.

Constraints may also be placed on the eventual design and construction of the product, as in the following example:

Constraints are simply another type of requirement.

The product shall operate as an iPad, iPhone, Android, and Blackberry app.

Providing that this is a real business constraint—and not just a matter of opinion—any solution that does not meet this constraint is clearly unacceptable.

Whatever they are constraining, constraints can be seen as another type of requirement. SeeFigure 1.4.

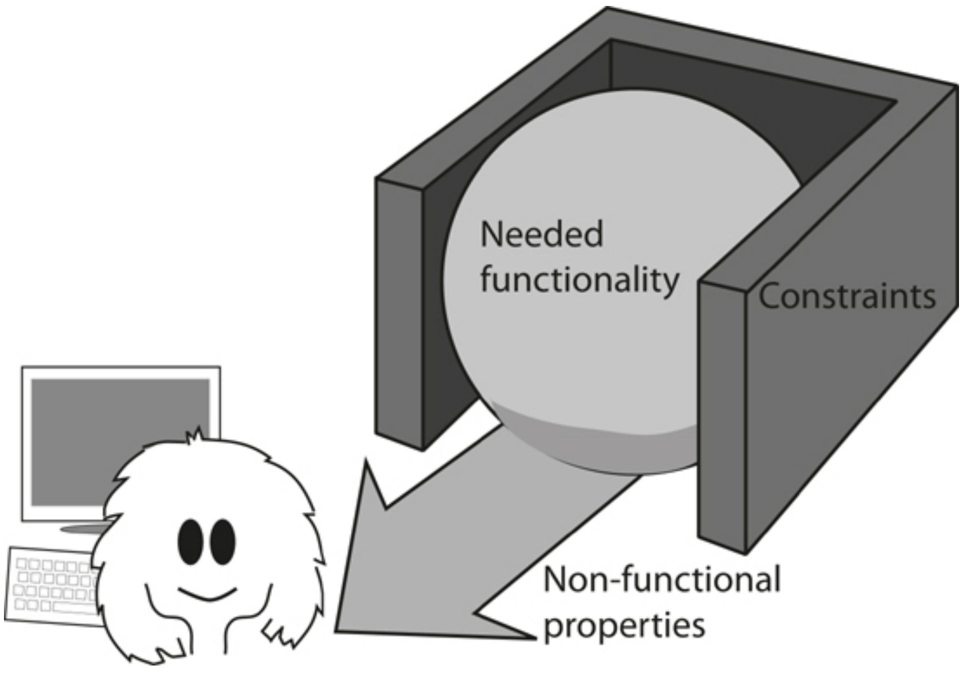


Figure 1.4. The functionality of the end product is restricted by the constraints. The functionality is to the benefit of its user, but it is the nonfunctional requirements that "deliver" the functionality by making the product usable and acceptable to the users.

Scope, Stakeholders, and Goals

Scope is not all there is to getting the requirements project off the ground. To build the right product, you have to understand the extent of the work; the people who do it, influence it, or know about it; and the outcome that those people are trying to achieve. This is the trinity of*scope, stakeholders, and goals,* as shown in Figure 3.6.

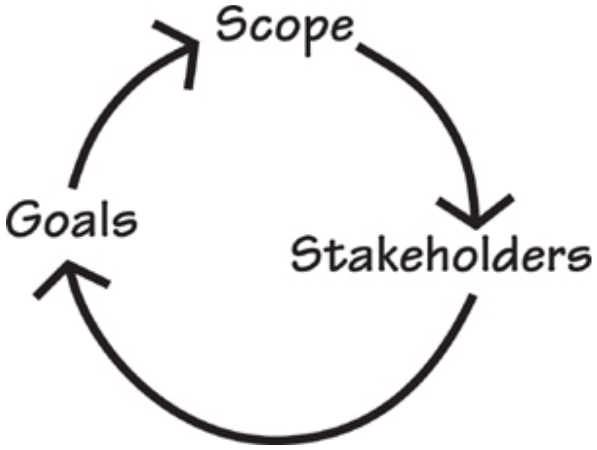


Figure 3.6. The scope, stakeholders, and goals are not decided in isolation from one another. Rather, the scope of the work indicates the stakeholders who have an interest in the work; the stakeholders, in turn, decide what they want the goals of the project to be.

The *scope* is the extent of the business area affected by the product. Because it is defining a part of a real-life organization, the scope points to the *stakeholders*—the people who have an interest in, or an effect on, the success of the work. The stakeholders, in turn, decide the*goals,* which is the improvement the business wants to experience when the product is installed.

There is no particular order to deciding what these factors should be. Most projects start with scope, but it is not obligatory—you use whatever information is to hand first. You have to iterate between the three factors until you have stabilized them, but this is almost always a short process when your organization knows why it wants to invest in the project.

Stakeholders

The next part of the trinity is the stakeholders. Stakeholders include anyone with an interest in, or an effect on, the outcome of the product. The owner is the most obvious stakeholder, but there are others. For example, the intended users of the product are stakeholders—they have an interest in having a product that does their work correctly. A subject-matter expert is an obvious stakeholder; a security expert is a less obvious stakeholder but one who must be considered for any product that could hold confidential or financial information. Potentially dozens of stakeholders exist for any project. Remember that you are trying to establish the optimal value for the owner, and that probably means talking to many people, all of them are potentially sources of requirements.

Stakeholders are the source of requirements.

The stakeholder map (Ian Alexander calls it an *onion diagram*) in Figure 3.7 identifies common classes of stakeholders that might be represented by one or more roles in your project. Let's look a little more closely at this map.

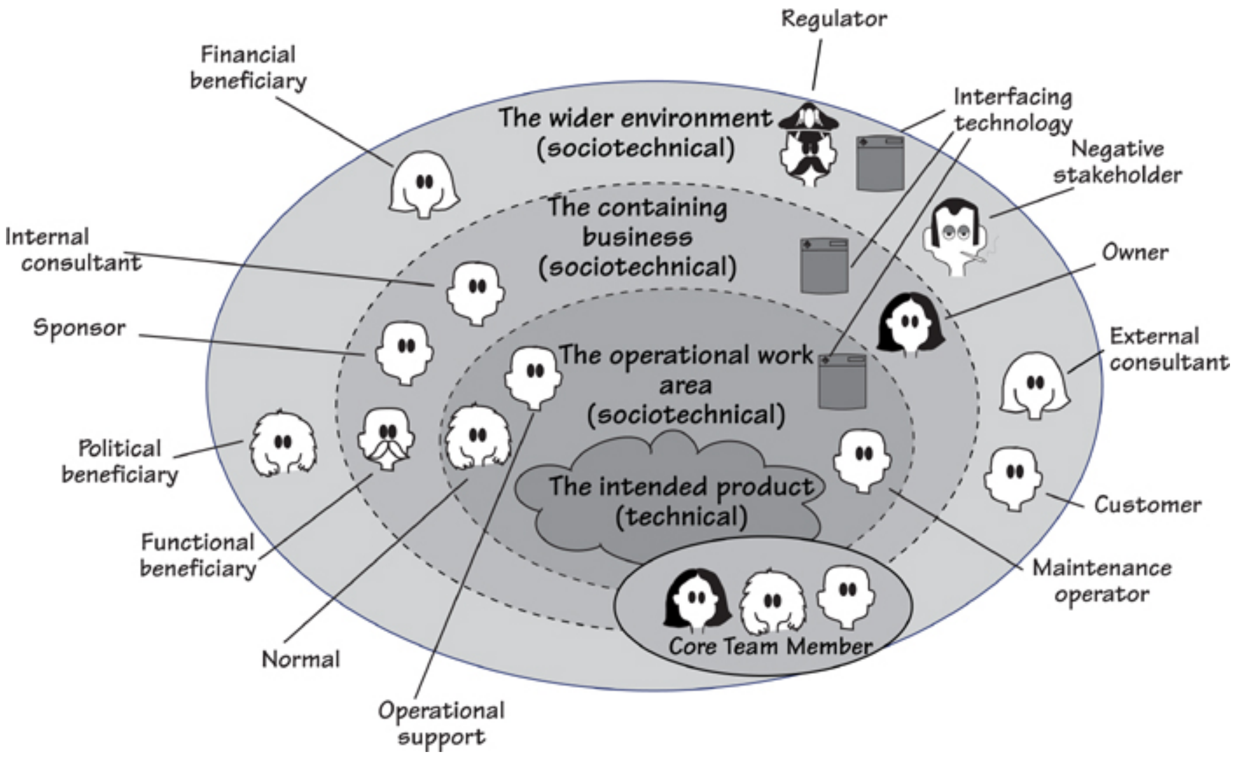


Figure 3.7. This stakeholder map shows the organizational rings surrounding the eventual product, and the classes of stakeholders who inhabit these rings. Use this map to help determine which classes of stakeholders are relevant to your project and which roles you need to represent them.

At the center of the stakeholder map is the *intended product*. Notice that it has a vague, cloud-like shape—this is intentional. At the start of the requirements activities you cannot be certain of the precise boundaries of the product, so for the moment we will leave it loosely defined. Surrounding the intended product is a ring representing the *operational work area*—stakeholders who will have some direct contact with the product inhabit this space. In the next ring, the *containing business,* you find stakeholders who benefit from the product in some way, even though they are not in the operational area. Finally, the outer ring, the *wider environment,* contains other stakeholders who have an influence on or an interest in the product. Note that the detailed and multiple involvement of the *core team members*—analysts, designers, project manager, and so on—is emphasized by the fact they span all the rings.

Reading

For more on stakeholder analysis, refer to Alexander, Ian, Neil Maiden, et al. *Scenarios, Stories, Use Cases Through the Systems Development Life-Cycle*. John Wiley & Sons, 2004.

Because so many classes of stakeholders exist, it is helpful to discuss some of the more important ones. After we have discussed these stakeholders, we will point you at a way of formalizing them with a stakeholder analysis template.

The Sponsor

We have said—often, and it shall be repeated—that the product has to provide the optimal value to its owner. However, for many products you do not, and usually cannot, have direct access to the owner. Many projects are carried out by commercial organizations, which are strictly speaking owned by their shareholders. Naturally enough, you can't go and talk to all the shareholders, nor are you likely to get access to the board of directors. In this case, the usual course of action is to appoint a sponsor for the project to represent the owner's interest; in many cases, the resources for the development of the new product that come from a sponsor's budget. You will find that the sponsor is concerned with project issues, and will be instrumental in setting some of the constraint requirements. We will deal with constraints in a little while.

The sponsor pays for the development of the product.

On the simple basis that "money talks," the sponsor, by paying for the development, has the final say in what that product does, how it does it, and how elaborate or how sparse it must be. In other words, the sponsor is the ultimate arbiter of which product will yield the optimal value.

You cannot proceed without a sponsor. If no one is representing the interest of the organization at large, then there is little point in proceeding with the project. The sponsor is most likely to be present at the blastoff meeting (you should be worried if the sponsor is not there) and is most likely one of these people:

• *User Management:* If you are building a product for in-house consumption, the most realistic sponsor is the manager of the users who will ultimately operate the product. Their department, or its work, is the beneficiary of the product, so it is reasonable that the cost of construction is borne by the departmental manager.

• *Marketing Department:* If you build products for sale to people outside your organization, the marketing department may assume the role of sponsor and represent the eventual owners of the product.

• *Product Development:* If you build software for sale, the budget for its development might be with your product manager or strategic program manager, in which case one of those would be the sponsor.

Consider your own organization, its structure and job responsibilities: Which people best represent the owner? Who pays for product development? Who, or what, reaps the benefit of the business advantage that the product brings? Whose values do you have to consider when you are determining what the end product is to do?

Do whatever you need to do to find your sponsor; your project will not succeed without one.

Let's assume the sponsor (in this case an owner representative) for the IceBreaker project is Mack Andrews, the CEO of Saltworks Systems. Mr. Andrews has made the commitment to invest in building the product. You record this agreement in Section 2 of your requirements specification:

The sponsor of the project is Mack Andrews, the CEO of Saltworks Systems. He has said that it is his goal to develop this product to appeal to a broader range of markets in other countries, including airports and their runways.

There are several things to note here. First, you name the sponsor. It is now clear to everyone on the project that Mack Andrews takes responsibility for investing in the product, and so will be the final arbiter on scope changes. Second, other information is provided about the sponsor that will be used as the project proceeds and may have a bearing on some of the requirements—particularly, the usability, adaptability, and "productization" requirements.

For more details on usability, adaptability, productization, and other types of requirements, refer to the template in Appendix A.

The Customer

The customer buys the product once it is developed, becoming the product's new owner. To persuade the customer to purchase the product, you have to build something that your customer will find valuable and useful and pleasurable.

The customer buys the product. You have to know this person well enough to understand what he finds valuable and, therefore, what he will buy.

Perhaps you already know the names of your customers, or perhaps they are hundreds or thousands of unknown people who might be persuaded to pay for your product. In either case, you have to understand them well enough to understand what they find valuable and what they will buy.

There is a difference between customers who buy for their own use and those who buy for use by others. When the product is a retail product and the customer and the owner are the same person, then that person's own values are of supreme importance to you. Is the customer looking for convenience? Most people are, and if so, you have to discover what the customer is doing that you can make more convenient. How much convenience will he find valuable?

When customers buy for use by others, the owner is presumably an organizational customer. Your interest lies in what the organization is doing, and what it considers valuable. That is, what can your product do that will make the users within the organization more productive, efficient, quicker, or whatever other quality it is seeking?

Even if you are developing open-source software, you still have a customer; the difference is simply that no money changes hands.

You must understand what appeals to your customers, and what they value. What will they find useful? What will they pay for? Understanding your customer correctly makes a huge difference to the success of your product.

For the IceBreaker product, the Northumberland County Highways Department has agreed to be the first customer.[1]

The customer for the product is the Northumberland County Highways Department, represented by director Jane Shaftoe.

As there is a single customer (at this stage), it would, of course, be advisable to invite her to participate as a stakeholder in the project. This kind of outreach results in the customer being actively involved in selecting which requirements are useful, choosing between conflicting requirements, and making the requirements analysts aware of her values, problems, and aspirations.

Saltworks Systems has further ambitions for the IceBreaker product. In the earlier statement about the sponsor, he said that he wanted an ice forecasting system that could be sold to road authorities and airports in other counties and other countries. If you plan to build the product with this aim, then your requirements specification should include an additional customer statement:

Potential customers for the product include all counties in the United Kingdom, northern parts of North America, and northern Europe and Scandinavia. A summary of the requirements specification will be presented to the Highways Department managers of selected counties, states, and countries for the purpose of discovering additional requirements.

It is clear that the customer must always be represented on the project. Where many potential customers exist, you must find a way of representing them in your project. This representation can come from the marketing department, senior users from one or more of your key customers, or a combination of domain and usability experts from within your organization. We will also later discuss personas as a way of representing the customers. The nature of your product, the structure of your organization, your customer base, and probably several other factors decide which roles within your organization are able to represent the customers.

Users: Understand Them

We are using the term *users* to mean the people who will ultimately be the hands-on operators of your product. The stakeholder map (Figure 3.7) refers to them as *normal operators, operational support,* and *maintenance operators*. For in-house products, the users are typically the people who work for the project sponsor. For personal computer or mobile device products, the user and the owner are often the same person.

The purpose of identifying the users is to understand what they are doing and which improvements they consider to be valuable.

Identifying your users is the first step in understanding the work they do—after all, your product is intended to improve this work. Additionally, you have to learn what kind of people they are so you provide the right user experience (we look at this issue later when we discuss usability requirements). You have to bring about a product that your users are both able to use and want to use. Obviously, the better your understanding of your users, the better the chance you have of specifying a suitable product for them.

Different users make different demands on your product. For example, an airline pilot has very different usability requirements from, say, a commuter buying a ticket on a rail system. If your user were a commuter, then a "person without cash" and a "person with only one arm free" would raise their own usability requirements.

Reading

Don Gause and Jerry Weinberg give a wonderful example of brainstorming lists of users in their book: Gause, Don, and Gerald Weinberg. *Exploring Requirements: Quality Before Design*. Dorset House, 1989. Although this text is getting a little long in the tooth, its advice is still relevant. Most of Jerry Weinberg's books are available as Kindle books or at Smashwords.

When developing consumer products, mass-market software, or websites, you should consider using a *persona* as the user. A persona is a virtual person who is archetypical of most of your users. By determining the characteristics of this persona to a sufficient degree, the requirements team can know the correct requirements to satisfy each of the personas. You should decide at this stage if you will use personas, but they can be more fully developed later. More information is provided on personas in Chapter 5, Investigating the Work.

A persona is a virtual person that is archetypical of most of your users.

The consideration of potential users is vital for agile development. Too many teams operate with only one user asked to supply the requirements for a product, and little or no consideration given to what will happen when the product is released to a wider audience. We strongly urge you to always consider the broadest spectrum of users and, at the very least, to choose user stakeholders from both ends of that spectrum.

Having identified your users, you have to record them. For example, in the Icebreaker project we had users in the following categories:

• Qualified Road Engineers

• Clerks in the truck department

• Managers

For each category of user, write a section in your specification to describe, as fully as time allows, the attributes of your users. Consider these possibilities:

• Subject-matter experience: How much help do they need?

• Technological experience: Can they operate the product? Which technical terms should be used?

• Intellectual abilities: Should tasks be made simpler? Or broken down to a lower level?

• Attitude toward the job: What are the users' aspirations?

• Education: What can you expect your user to know?

• Linguistic skills: Not all your users will speak or read the home language.

• And most importantly, what is it about their work that they most wish to improve?

Also, for each category of user, identify the particular attributes that your product must cater to:

• People with disabilities: Consider all disabilities. This, in some cases, is a legal requirement.

• Nonreaders: Consider people who cannot read and people who do not speak the home language.

• People who need reading glasses: This is particularly near and dear to one of the authors.

• People who cannot resist changing things like fonts, styles, and so on.

• People who will probably be carrying luggage, large parcels, or a baby.

• People who do not normally use a computer.

• People who might be angry, frustrated, under pressure, or in a hurry.

We realize that writing down all this stuff seems like a chore. However, we have found that taking the time to record it so other people can read it is one of the few ways for you to demonstrate that you understand it. The users are so important to your cause that you must understand what kind of people they are and which capabilities they have. To wave your hands and say, “They are graphic designers” or “They want to buy books on the Web,” falls short of the minimum level of understanding.

Another category of stakeholder in the operational work area is the maintenance operator. Your product is likely to have maintainability requirements, and you can learn about them from this person.

Operational support is another source of requirements relating to the operational work area. Roles that are sources of these requirements include help desk staff, trainers, installers, and coaches.

People other than the intended users might end up having contact with your product. It is better to identify superfluous users than to fail to find them all.

At this stage, any users you identify are *potential* users. That is, you do not yet precisely know the scope of the product—you determine this later in the requirements process—so you are identifying the people who might possibly use, maintain, and support the product. Remember that people other than the intended users (e.g., firefighters, security personnel) might end up having hands-on contact with your product. It is better to identify superfluous users than to fail to find them all because each different category of user will have some different requirements.

Other Stakeholders

There are more people you have to talk to so that you can find all the requirements. Your contact with these people might be fleeting, but it is nevertheless necessary. Any of them may potentially have requirements for your product.

“Every context is composed of individuals who do or do not decide to connect the fate of a project with the fate of the small or large ambitions they represent.”

—Bruno Latour, *ARAMIS or the Love of Technology*

The problem that you often face in requirements projects is that you fail to discover all the stakeholders, and thus fail to discover their needs. This shortcoming may result in a string of change requests when the product is released to an audience that is wider than at first thought. Naturally, the people who were overlooked will not be happy. Additionally, you must consider that when any new system is installed, someone gains and someone loses power: Some people find that the product brings them new capabilities, and some people are not able to do their jobs the way they used to do them. The moral of the story is clear: Find all parties who will be affected by the product, and find their requirements.

We list our stakeholders in Section 2 of the Volere Requirements Specification Template. You can find this template in Appendix A. The list acts as a checklist for finding the appropriate stakeholders.

Let's consider some other stakeholders by looking at some candidate categories. You can also see most of these groups illustrated as classes on the stakeholder map shown in Figure 3.7.

Consultants

Consultants—both internal to your organization and external—are people who have expertise you need. Consultants might never touch or see your product, but their knowledge becomes part of it. For example, if you are building a financial product, a security expert is one of your stakeholders. He might never see your product, but his expertise—and this is his stake in the requirements—ensures the product is secure.

Management

Consider any category of management. These groups show up on the stakeholder map (Figure 3.7) as classes like *functional beneficiary, political beneficiary,* and *financial beneficiary.* Is it a strategic product? Do any managers other than those directly involved have a stake?

Product managers and program managers are obvious sources of requirements. Project managers or leaders who are responsible for the day-to-day management of the project effort likewise have contributions to make.

Subject-Matter Experts

This constituency may include domain analysts, business consultants, business analysts, or anyone else who has some specialized knowledge of the business subject. As a consequence, these experts are a prime source of information about the work.

Core Team

The core team is made up of the people who are part of the building effort for the product. They may include product designers, developers, testers, business analysts, systems analysts, systems architects, technical writers, database designers, and anyone else who is involved in the construction.

You can also consider the open-source community as stakeholders—they have knowledge about technology and trends in most areas of software. You can contact these people via open-source forums. They are usually very enthusiastic and ready to share knowledge with you.

When you know the people involved, record their names. Otherwise, use this section of the template to list the skills and duties that you think are most likely needed to build the product.

Inspectors

Consider auditors, government inspectors, any kind of safety inspectors, technical inspectors, or even possibly the police. It may well be necessary to build inspection capabilities into your product. If your product is subject to the Sarbanes-Oxley Act, or any of the other regulatory acts, then inspection is crucial, as are the requirements from the inspectors.

Market Forces

People from the marketing department are probably the stakeholders representing the marketplace. When you are building a product for commercial sale, trends in the market are a potent source of requirements, as is a deep knowledge of the likely consumers. Note the speed at which the smart phone and tablet markets are moving (at the time of writing). Staying ahead of the curve is vital for any consumer product.

Legal Experts

Each year the world is filled with more and more laws—complying with all of them is daunting but necessary. Your lawyers are the stakeholders for most of your legal requirements.

Negative Stakeholders

Negative stakeholders are people who do not want the project to succeed (remember what we said previously about losing power). Although they may not be the most cooperative individuals, you would be wise to consider them. You may find that, if their requirements are different from the commonly perceived version, and you can accommodate their requirements, your opposition could well become your supporters.

You might also consider the people who threaten your product—the hackers, defrauders, and other malevolent people. You will get no cooperation from them, but you should consider how they might mistreat your product.

Industry Standard Setters

Your industry may have professional bodies that expect certain codes of conduct to be followed or certain standards to be maintained by any product built within the industry or for use by the industry.

Public Opinion

Do any user groups for your product exist? They will certainly be a major source of requirements. For any product intended for the public domain, consider polling members of the public about their opinion. They may make demands on your product that could spell the difference between acceptance and rejection.

Government

Some products must interact with government agencies for reporting purposes, or receive information from a government agency; other products have requirements that necessitate consulting with the government. Although the government may not assign a person full-time to your project, you should nevertheless nominate the pertinent agency as a stakeholder.

Special-Interest Groups

Consider handicapped-interest groups, environmental bodies, foreign people, old people, gender-related interests, or almost any other group that may come in contact with your product.

Technical Experts

Technical experts do not necessarily build the product, but they will almost certainly be consulted about some part of it. For the stakeholders from this constituency, consider usability experts, security consultants, hardware people, experts in the technologies that you might use, specialists in software products, or experts from any technical field that the product could use.

Cultural Interests

This constituency is applicable to products intended for the public domain, and especially when your product is to be sold or seen in other countries. In addition, it is always possible in these politically correct times that your product could offend someone. If there is any possibility that religious, ethnic, cultural, political, gender, or other human interests could be affected by or come into contact with your product, then you should consider representatives from these groups as stakeholders for the project.

Adjacent Systems

The adjacent systems on your work context diagram are the systems, people, or work areas that directly interact with the work you are studying. Look at each adjacent system: Who represents its interests, or who has knowledge of it? When the adjacent system is an automated one, who is its project leader or maintainer? If these stakeholders are not available, you might have to read the adjacent system's documentation, or its code, to discover whether it has any special demands for interacting with your product. For each adjacent system you need to find at least one stakeholder.

Finding the Stakeholders

At scoping time, you normally inspect your context model and hold a brainstorming session to identify all possible stakeholders. You do not have to start from scratch; we have constructed a spreadsheet with many categories of stakeholders, along with the kind of knowledge you need to obtain from each person. This spreadsheet (see Appendix

B) cross-references the stakeholder map (Figure 3.7) and provides a detailed specification of your project's sociology. Once you have identified the stakeholder, add that person's name to the list. The complete spreadsheet is available as a free download at www.volere.co.uk.

The greatest problem concerning stakeholders is the requirements you miss when you don't find all of the stakeholders.

You will be talking to the stakeholders, so at this stage it pays to explain to them why they are stakeholders and why you need to consult them about requirements for the product. Explain specifically why their input will make a difference to the eventual product. It is polite to inform stakeholders of the amount of their time you require and the type of participation that you have in mind; a little warning always helps them to think about their requirements for the product. The greatest problem concerning stakeholders is the requirements that you miss if you do not find all of the stakeholders, or if you exclude stakeholders from the requirements-gathering process.

See the Stakeholder Management Template in Appendix B, also available as a downloadable Excel spreadsheet at www.volere.co.uk.

Chapter 6. User Stories

With Pete Behrens

They have been at a great feast of languages, and stol'n the scraps.

—Shakespeare, *Love's Labour's Lost*, Act 5, scene 1

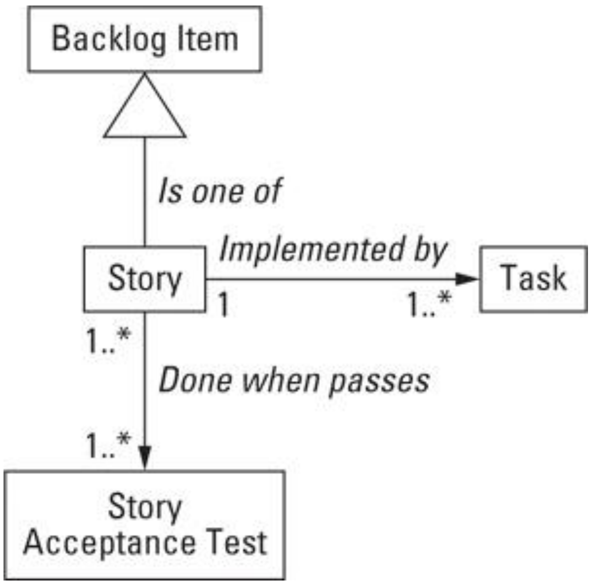
Introduction

In Chapter 3, Agile Requirements for the Team, we introduced the concepts and relationships among the key artifacts—backlogs, user stories, tasks, and so on—used by agile teams to define, build, and test the system of interest. We noted that the user story is the workhorse of agile development, and it is the container that carries the value stream to the user. It also serves as a metaphor for our entire incremental value delivery approach, that is:

Define a user value story, implement and test it in a short iteration, demonstrate/and or deliver it to the user, repeat forever!

We summarize the requirements artifacts involved in fulfilling this mission in Figure 6-1.

Figure 6-1. Requirements model for teams



From the figure, we see that stories come from the backlog and are implemented via whatever design, coding, and testing tasks are needed to complete the story. Further, stories cannot be considered to be *done* until they pass an associated acceptance test.

In this chapter, we'll describe the user story in more detail, because it is there that we will find the agile practices that help us conform our solution directly to the user's specific needs and help assure quality at the same time.

User Story Overview

We have noted many of the contributions of Scrum to enterprise agile practices, including, for example, the definition of the product owner role, which is integral to our requirements practices. But it is to XP that we owe the invention of the user story, and it is the proponents of XP who have developed the breadth and depth of this artifact. As Beck and Fowler [2005] explain:

The story is the unit of functionality in an XP *project. We demonstrate progress by delivering tested, integrated code that implements a story. A story should be understandable to customers, developer-testable, valuable to the customer, and small enough that the programmers can build half a dozen in an iteration.*

However, though the user story originated in XP, this is less of a "methodological fork in the road" than it might appear, because user stories are now routinely taught within the constructs of Scrum training as a tool for building product backlogs and defining Sprint content. We have Mike Cohn to thank for much of this integration; he has developed user stories extensively in his book *User Stories Applied* [Cohn 2004], and he has been very active in the Scrum community.

For our purposes, we'll define a user story simply as follows:

A user story is a brief statement of intent that describes something the system needs to do for the user.

In XP, user stories are often written by the customer, thus integrating the customer directly in the development process. In Scrum, the product owner often writes the user stories, with input from the customers, the stakeholders, and the team. However, in actual practice, any team member with sufficient domain knowledge can write user stories, but it is up to the product owner to accept and prioritize these potential stories into the product backlog.

User stories are a tool for defining a system's behavior in a way that is understandable to both the developers and the users. User stories focus the work on the value defined by the user rather than a functional breakdown structure, which is the way work has traditionally been tasked. They provide a lightweight and effective approach to managing requirements for a system.

A user story captures a short statement of function on an index card or perhaps with an online tool. In simple backlog form, stories can just be a list of things the system needs to do for the user. Here's an example:

Log in to my web energy-monitoring portal.

See my daily energy usage.

Check my current electricity billing rate.

Details of system behavior do not appear in the brief statement; these are left to be developed later through conversations and acceptance criteria between the team and the product owner.

User Stories Help Bridge the Developer–Customer Communication Gap

In agile development, it is the developer's job to speak the language of the user, not the user's job to speak the language of developers. Effective communication is the key, and we need a common language. The user story provides the common language to build understanding between the user and the technical team.

Bill Wake, one of the creators of XP, describes it this way:[1]

A pidgin language is a simplified language, usually used for trade that allows people who can't communicate in their native language to nonetheless work together. User stories act like this. We don't expect customers or users to view the system the same way that programmers do; stories act as a pidgin language where both sides can agree enough to work together effectively.

With user stories, we don't have to understand each other's language with the degree of proficiency necessary to craft a sonnet; we just need to understand each other enough to know when we have struck a proper bargain!

User Stories Are Not Requirements

Although user stories do most of the work previously done by software requirements specifications, use cases, and the like, they are *materially different* in a number of subtle yet critical ways.

• They are not detailed requirements specifications (something a system shall do) but are rather negotiable expressions of intent (it needs to do something about like this).

• They are short, easy to read, and understandable to developers, stakeholders, and users.

• They represent small increments of valued functionality that can be developed in a period of days to weeks.

• They are relatively easy to estimate, so effort to implement the functionality can be rapidly determined.

• They are not carried in large, unwieldy documents but rather organized in lists that can be more easily arranged and rearranged as new information is discovered.

• They are not detailed at the outset of the project but are elaborated on a just-in-time basis, thereby avoiding too-early specificity, delays in development, requirements inventory, and an over-constrained statement of the solution

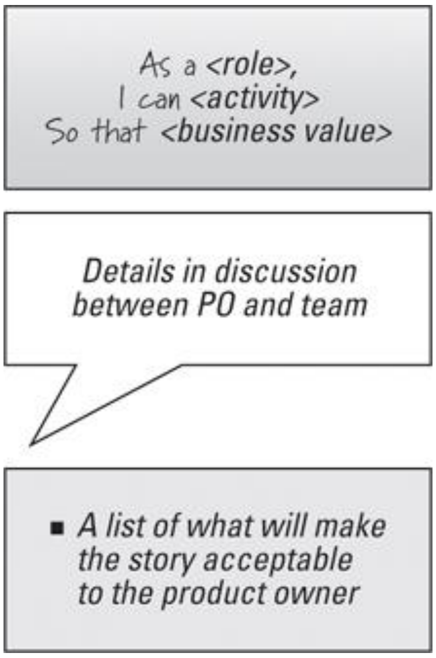
• They need little or no maintenance and can be safely discarded after implementation.[2],[3]

• User stories, and the code that is created quickly thereafter, serve as inputs to documentation, which is then developed incrementally as well.

User Story Form

This section addresses formats for user stories.

Card, Conversation, and Confirmation


As a <role>,
I can <activity>
So that <business value>
Details in discussion between PO and team
A list of what will make the story acceptable to the product owner

Ron Jeffries, another creator of XP, described what has become our favorite way to think about user stories. He used the alliteration *card, conversation, and confirmation* to describe the three elements of a user story.[4]

Card represents two to three sentences used to describe the intent of the story. The card serves as a memorable token, which summarizes intent and represents a more detailed requirement, whose details remain to be determined.

Note

In XP and agile, stories are often written manually on physical index cards. More typically in the enterprise, the “card” element is captured as text and attachments in a spreadsheet or agile project management tooling, but teams often still use cards for early planning and brainstorming, as we will see later.

Conversation represents a discussion between the team, customer, product owner, and other stakeholders, which is necessary to determine the more detailed behavior required to implement the intent. In other words, the *card* also represents a “promise for a conversation” about the intent.

Confirmation represents the *acceptance test*, which is how the customer or product owner will confirm that the story has been implemented to their satisfaction. In other words, confirmation represents the *conditions of satisfaction* that will be applied to determine whether the story fulfills the intent as well as the more detailed requirements.

With this simple alliteration, we have an object lesson in how quality in agile is achieved during, rather than after, actual code development. We do that by simply making sure that every new user story is discussed and refined in whatever detail is necessary and is tested to the satisfaction of the key stakeholders.

User Story Voice

In the last few years, a newer, fairly standardized form has been applied that strengthens the user story construct significantly. The form is as follows:

As a <role>*, I can <activity> so that* <business value>.

where:

• <role> represents who is performing the action or perhaps one who is receiving the value from the activity. It may even be another system, if that is what is initiating the activity.
• <activity> represents the action to be performed by the system.
• <business value> represents the value achieved by the activity.

We call this the *user voice* form of user story expression and find it an exceedingly useful construct[5] because it spans the problem space (<business value> delivered) and the solution space (<activity> the user performs with the system). It also provides a user-first (<role>) perspective to the team, which keeps them focused on business value and solving real problems for real people.

This user story form greatly enhances the “why” and “how” understanding that developers need to implement a system that truly meets the needs of the users.

For example, a user of a home energy-management system might want to do the following:

As a Consumer (<role>), I want to be able to see my daily energy usage (<what I do with the system>) so that I can lower my energy costs and usage (<business value I receive>).”

Each element provides important expansionary context. The *role* allows a segmentation of the product functionality and typically draws out other role-based needs and context for the activity. The *activity* typically represents the “system requirement” needed by the role. And the *value* communicates why the activity is needed, which can often lead the team to finding possible alternative activities that could provide the same value for less effort.

User Story Detail

The details for user stories are conveyed primarily through conversations between the product owner and the team, keeping the team involved from the outset. However, if more details are needed about the story, they can be provided in the form of an attachment (mock-up, spreadsheet, algorithm, or whatever), which is attached to the user story. In that case, the user story serves as the “token” that also carries the more specific behavior to the team. The additional user story detail should be collected over time (just-in-time) through discussions and collaboration with the team and other stakeholders before and during development.

User Story Acceptance Criteria

In addition to the statement of the user story, additional notes, assumptions, and acceptance criteria can be kept with a user story. *Many* discussions about a story between the team and customers will likely take place while the story is being coded. The alternate flows in the activity, acceptance boundaries, and other clarifications should be captured along with the story. Many of these can be turned into acceptance test cases, or other functional test cases, for the story.

Here’s an example:

As a consumer, I want to be able to see my daily energy usage so that I can lower my energy costs and usage.

Acceptance Criteria:

• Read DecaWatt meter data every 10 seconds and display on portal in 15-minute increments and display on in-home display every read.

• Read KiloWatt meters for new data as available and display on the portal every hour and on the in-home display after every read.

• No multiday trending for now (another story).

Etc . . . .

Acceptance criteria are not functional or unit tests; rather, they are the conditions of satisfaction being placed on the system. Functional and unit tests go much deeper in testing all functional flows, exception flows, boundary conditions, and related functionality associated with the story.

INVEST in Good User Stories

Agile teams spend a significant amount of time in discovering, elaborating, and understanding user stories and writing acceptance tests for them. This is as it should be, because it represents the following conclusion:

Writing the code for an understood objective is not necessarily the hardest part of software development; rather, it is understanding what the real objective for the code *is*.

Therefore, *investing* in good user stories, albeit at the last responsible moment, is a worthy effort for the team. Bill Wake coined the acronym INVEST[6] to describe the attributes of a good user story.

Independent

Negotiable

Valuable

Estimable

Small

Testable

The INVEST model is now fairly ubiquitous, and many agile teams evaluate their stories with respect to these attributes. Here's our view of the value of the team's INVESTment.

Independent

Independence means that a story can be developed, tested, and potentially even delivered on its own. Therefore, it can also be independently *valued*.

Many stories will have some natural sequential dependencies as the product functionality builds, and yet each piece can deliver value independently. For example, a product might display a single record and then a list, then sort the list, filter the list, prepare a multipage list, export the list, edit items in the list, and so on. Many of these items have sequential dependencies, yet each item provides independent value, and the product can be potentially shipped through any stopping point of development.

However, many nonvalued dependencies, either technical or functional, also tend to find their way into backlogs, and these we need to find and eliminate. For example, the following might be a nonvalued functional dependency:

As an administrator, I can set the consumer's password security rules so that users are required to create and retain secure passwords, keeping the system secure.

As a consumer, I am required to follow the password security rules set by the administrator so that I can maintain high security to my account.

In this example, the consumer story depends on the administrator story. The administrator story is testable only in setting, clearing, and preserving the policy, but it is not testable as enforced on the consumer. In addition, completing the administrator story does not leave the product in a potentially shippable state—therefore, it's not independently valuable.

By reconsidering the stories (and the design of the system), we can remove the dependency by splitting the stories in a different manner, in this case through the types of security policies applied and by combining the setup with enforcement in each story:

As an administrator, I can set the password expiration period so that users are forced to change their passwords periodically.

As an administrator, I can set the password strength characteristics so that users are required to create difficult-to-hack passwords.

Now, each story can stand on its own and can be developed, tested, and delivered independently.

Negotiable . . . and Negotiated

Unlike traditional requirements, a user story is not a contract for specific functionality but rather a placeholder for requirements to be discussed, developed, tested, and accepted. This process of negotiation between the business and the team recognizes the legitimacy and primacy of the business inputs but allows for discovery through collaboration and feedback.

In our prior, siloed organizations, written requirements were generally required to facilitate the limited communication bandwidth between departments and to serve as a record of past agreements. Agile, however, is founded on the concept that a team-based approach is more effective at solving problems in a dynamic collaborative environment. A user story is real-time and structured to leverage this effective and direct communication and collaboration approach.

Finally, the negotiability of user stories helps teams achieve predictability. The lack of overly constraining and too-detailed requirements enhances the team's and business's ability to make trade-offs between functionality and delivery dates. Because each story has flexibility, the team has more flexibility to meet release objectives, which increases dependability and fosters trust.

Valuable

An agile team's goal is simple: to deliver the most value given their existing time and resource constraints. Therefore, value is the most important attribute in the INVEST model, and every user story must provide some value to the user, customer, or stakeholder of the product. Backlogs are prioritized by value, and businesses succeed or fail based on the value the teams can deliver.

A typical challenge facing teams is learning how to write small, incremental user stories that can effectively deliver value. Traditional approaches have taught us to create functional breakdown structures based on technical components. This technical layering approach to building software delays the value delivery until all the layers are brought together after multiple iterations. Wake[7] provides his perspective of vertical, rather than technical, layering:

Think of a whole story as a multi-layer cake, e.g., a network layer, a persistence layer, a logic layer, and a presentation layer. When we split a story [horizontally], we're serving up only part of that cake. We want to give the customer the essence of the whole cake, and the best way is to slice vertically through the layers. Developers often have an inclination to work on only one layer at a time (and get it "right"); but a full database layer (for example) has little value to the customer if there's no presentation layer.

Creating valuable stories requires us to reorient our functional breakdown structures from a horizontal to a vertical approach. We create stories that slice through the architecture so that we can present value to the user and seek their feedback as early and often as possible.

Although normally the value is focused on the user interacting with the system, sometimes the value is more appropriately focused on a customer representative or key stakeholder. For example, perhaps a marketing director is requesting a higher click-through rate on ads presented on the Web site. Although the story could be written from the perspective of the end user . . .

As a consumer, I can see other energy pricing programs that appeal to me so that I can enroll in a program that better suits my lifestyle.

. . . to provide a clearer perspective on the real value, it would be more appropriately written from the marketing director's perspective:

As a utility marketing director, I can present users with new pricing programs so that they are more likely to continue purchasing energy from me.

Another challenge faced by teams is to articulate value from technical stories such as code refactoring, component upgrades, and so on. For example, how would the product owner determine the value of the following?

Refactor the error logging system.

Articulating the value of a technical solution as a user story will help communicate to the business its relative importance. Here's an example:

As a consumer, I can receive a consistent and clear error message anywhere in the product so that I know how to address the issue. OR

As a technical support member, I want the user to receive a consistent and clear message anywhere in the application so they can fix the issue without calling support.

In these latter examples, the value is clear to the user, to the product owner, to the stakeholders, and to the team.

Estimable

A good user story is estimable. Although a story of any size can be in the backlog, in order for it to be developed and tested in an iteration, the team should be able to provide an approximate estimation of its complexity and amount of work required to complete it. The minimal investment in estimation is to determine whether it can be completed within a single iteration. Additional estimation accuracy will increase the team's predictability.

If the team is unable to estimate a user story, it generally indicates that the story is too large or uncertain. If it is *too large* to estimate, it should be split into smaller stories. If the story is*too uncertain* to estimate, then a technical or functional spike story can be used to reduce uncertainty so that one or more estimable user stories result. (Each of these topics is discussed in more detail in the following sections.)

One of the primary benefits of estimating user stories is not simply to derive a precise size but rather to draw out any hidden assumptions and missing acceptance criteria and to clarify the team's shared understanding of the story. Thus, the conversation surrounding the estimation process is as (or more) important than the actual estimate. The ability to estimate a user story is highly influenced by the size of the story, as we'll see shortly.

Small

User stories should be small enough to be able to be completed in an iteration. Otherwise, they can't provide any value or be considered *done* at that point. However, even smaller user stories provide more agility and productivity. There are two primary reasons for this:*increased throughput* and *decreased complexity*.

Increased Throughput

From queuing theory, we know that smaller batch sizes go through a system faster. This is one of the primary principles of lean flow and is captured in Little's law:

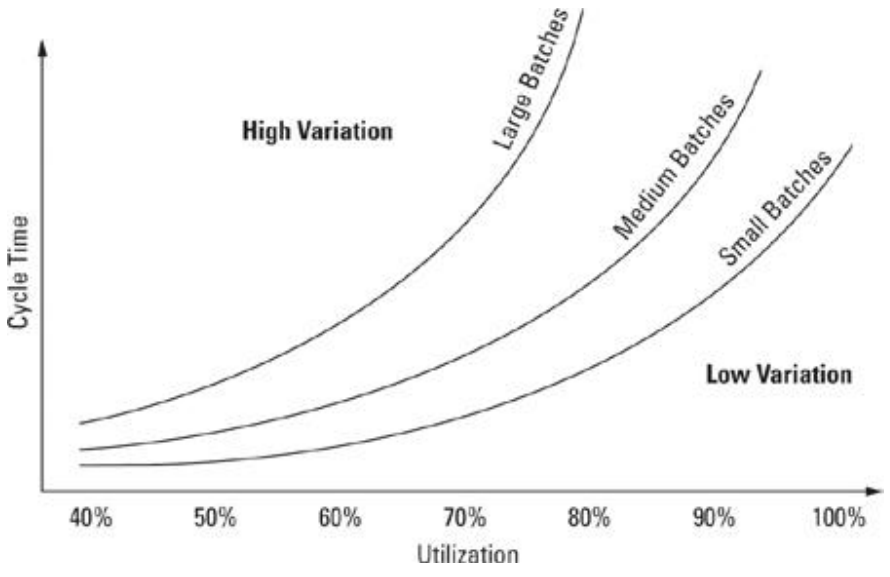
$$\text{Cycle Time} = \frac{\text{Work In Process}}{\text{Throughput}}$$

In a stable system (where throughput, the amount of work that can be done in a unit of time, is constant), we have to decrease work in process (the amount of things we are working on) in order to decrease cycle time (the time elapsed between the beginning and end of the process). In our case, that means*fewer, smaller stories in process will come out faster*.

Moreover, when a system is loaded to capacity, it can become unstable, and the problem is compounded. In heavily loaded systems, larger batches move disproportionately slower (throughput decreases) through the system. (Think of a highway system at rush hour. Motorcycles and bicycles have a much higher throughput than do cars and trucks. There is more space to maneuver smaller things through a loaded system.) Because development teams are typically fully allocated at or above capacity (80% to 120%), they fall in the "rush-hour highway" category.

When utilization hits 80% or so, larger objects increase cycle time (slow down) much more than smaller objects. Worse, the *variation* in cycle time increases, meaning that it becomes harder to predict when a batch might actually exit the system, as shown in Figure 6-2. In turn, this lower predictability wreaks havoc with schedules, commitments, and the credibility of the team.

Figure 6-2. Large batches have higher cycle times and higher cycle time variability [Poppendieck and Poppendieck 2007].



Decreased Complexity

Smaller stories not only go through faster because of their raw, proportional size, but they go through faster yet because of their *decreased complexity*, and complexity has a *nonlinear relationship* to size. This is seen most readily in testing, where the permutations of tests required to validate the functionality increase at an exponential rate with the complexity of the function itself. This correlates to the advice we receive about developing clean code, as Robert Martin [2009] notes on his rules for writing software functions.

- Rule 1: Do one thing.
- Rule 2: Keep them small.
- Rule 3: Make them smaller than that.

This is one of the primary reasons that the Fibonacci estimating sequence (that is, 1, 2, 3, 5, 8, 13, 21 . . .) is so effective in estimating user stories. The effort estimate grows nonlinearly with increasing story size.

On the Relationship of Size and Independence

A fair question arises as to the relationship between size and independence, because it seems logical that smaller stories increase the number of dependencies. However, smaller stories, even with some increased dependency, deliver higher value throughput and provide faster user feedback than larger stories. So, the agilist always leans to smaller stories *and then makes them smaller still*.

Testable

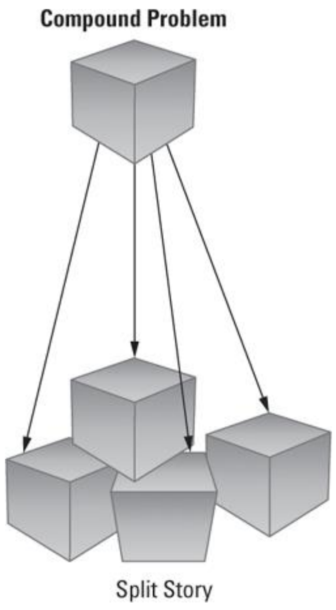
In proper agile, *all code is tested code*, so it follows that stories must be testable. If a story does not appear to be testable, then the story is probably ill-formed, overly complex, or perhaps dependent on other stories in the backlog.

To assure that *stories don't get into an iteration if they can't get out* (be successfully tested), many agile teams today take a “write-the-test-first” approach. This started in the XP community using test-driven development, a practice of writing automated unit tests prior to writing the code to pass the test.

Since then, this philosophy of approach is being applied to development of story acceptance criteria and the necessary functional tests prior to coding the story itself. If a team really knows how to test a story, then they likely know how to code it as well.

To assure testability, user stories share some common testability pitfalls with requirements. Vague words such as *quickly*, *manage*, *nice*, *clean*, and so on, are easy to write but very difficult to test because they mean different things to different people and therefore should be avoided. And although these words do provide negotiability, framing them with some clear boundaries will help the team and the business share expectations of the output and avoid big surprises.

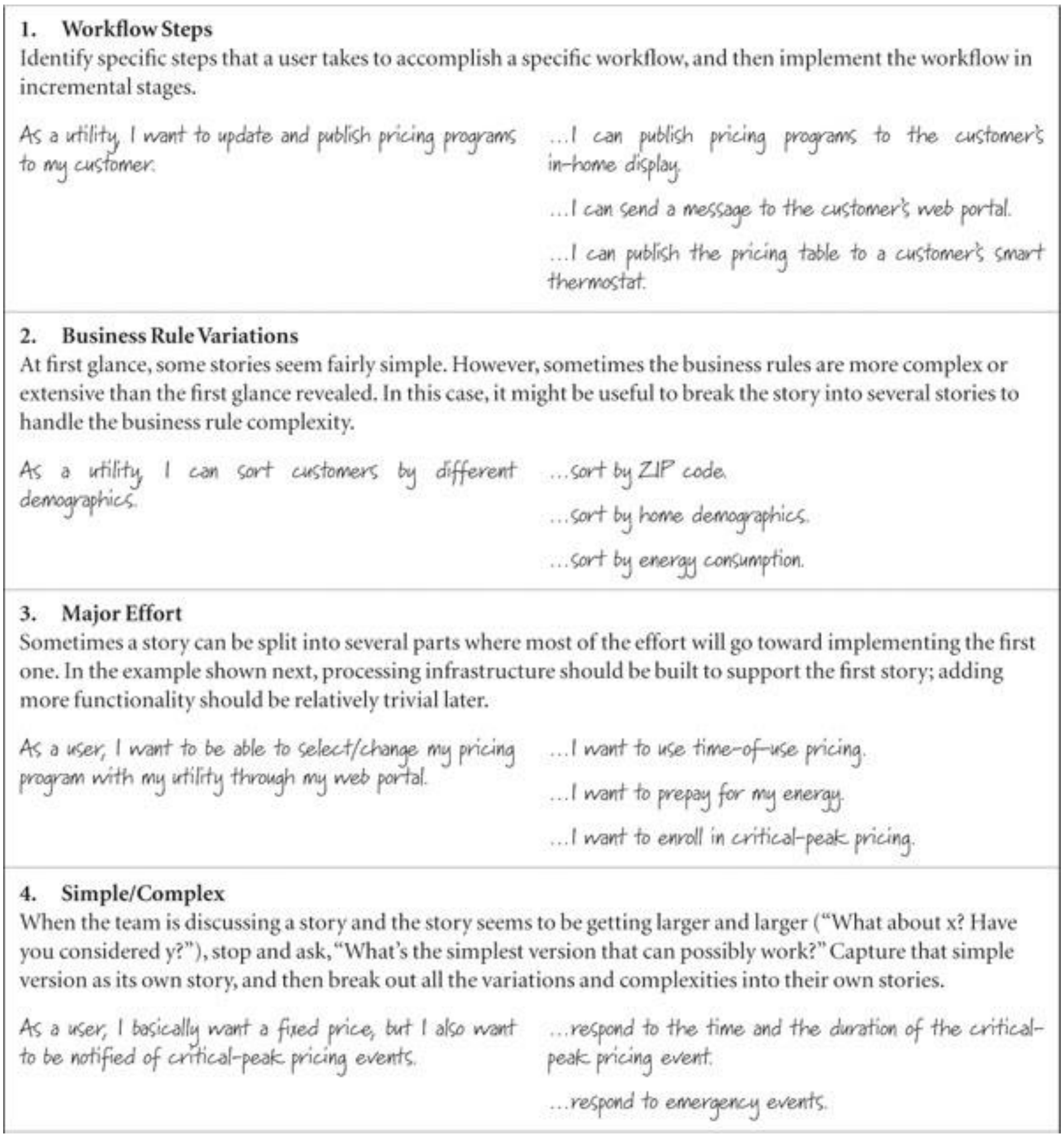
Splitting User Stories



User stories are often driven by epics and features—a large, vague concept of something we want to do for a user. We often find these big-value stories during our discovery process and capture them in the backlog. However, these are *compound stories*, as pictured on the left, and are usually far too big to be implemented within an iteration. To prepare the work for iterations, a team must break them down into smaller stories.

There is no set routine for splitting user stories into iteration-sized bites, other than the general guidance to make each story provide a vertical slice, some piece of user value, through the system. However, we recommend applying an appropriate selection of *ten common patterns to split a user story*, as Table 6-1 indicates.[8]

Table 6-1. Ten Patterns for Splitting a User Story



1. Workflow Steps
Identify specific steps that a user takes to accomplish a specific workflow, and then implement the workflow in incremental stages.

As a utility, I want to update and publish pricing programs to my customer.	…I can publish pricing programs to the customer's in-home display.
	…I can send a message to the customer's web portal.
	…I can publish the pricing table to a customer's smart thermostat.

2. Business Rule Variations
At first glance, some stories seem fairly simple. However, sometimes the business rules are more complex or extensive than the first glance revealed. In this case, it might be useful to break the story into several stories to handle the business rule complexity.

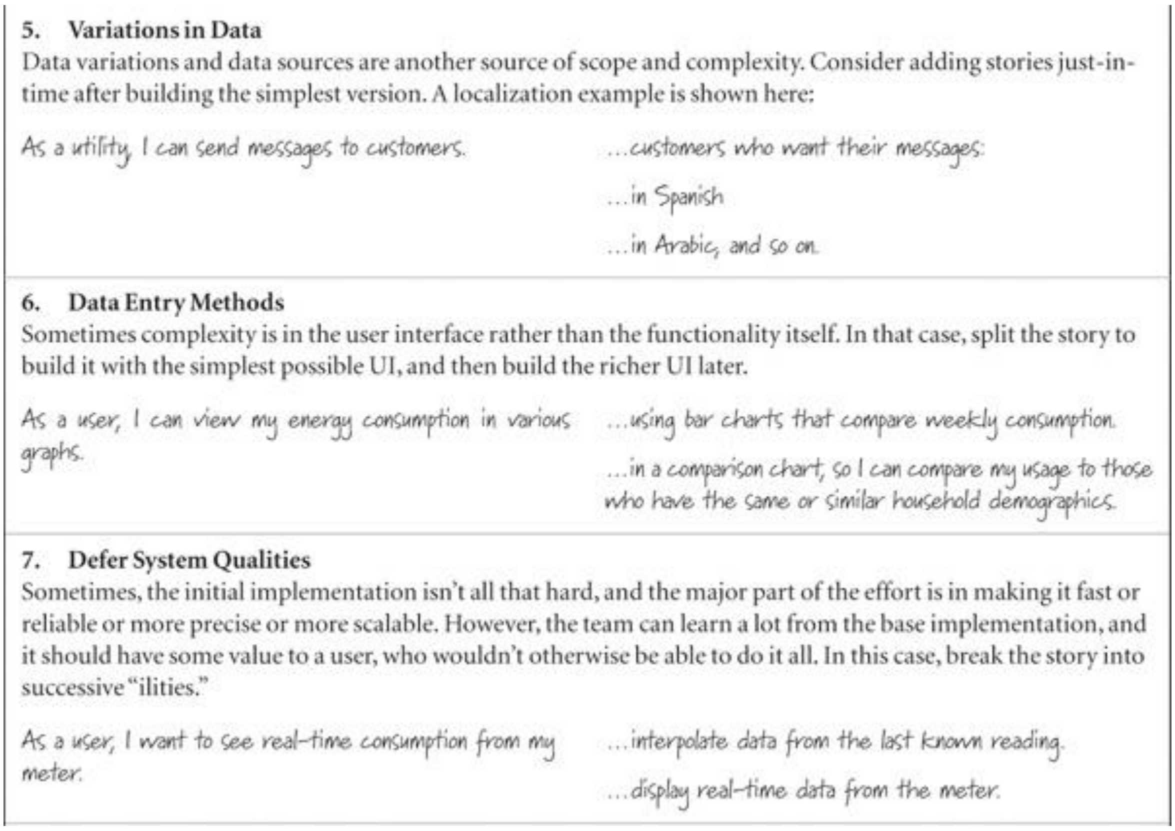
As a utility, I can sort customers by different demographics.	…sort by ZIP code.
	…sort by home demographics.
	…sort by energy consumption.

3. Major Effort
Sometimes a story can be split into several parts where most of the effort will go toward implementing the first one. In the example shown next, processing infrastructure should be built to support the first story; adding more functionality should be relatively trivial later.

As a user, I want to be able to select/change my pricing program with my utility through my web portal.	…I want to use time-of-use pricing.
	…I want to prepay for my energy.
	…I want to enroll in critical-peak pricing.

4. Simple/Complex
When the team is discussing a story and the story seems to be getting larger and larger ("What about x? Have you considered y?"), stop and ask, "What's the simplest version that can possibly work?" Capture that simple version as its own story, and then break out all the variations and complexities into their own stories.

As a user, I basically want a fixed price, but I also want to be notified of critical-peak pricing events.	…respond to the time and the duration of the critical-peak pricing event.
	…respond to emergency events.



5. Variations in Data
Data variations and data sources are another source of scope and complexity. Consider adding stories just-in-time after building the simplest version. A localization example is shown here:

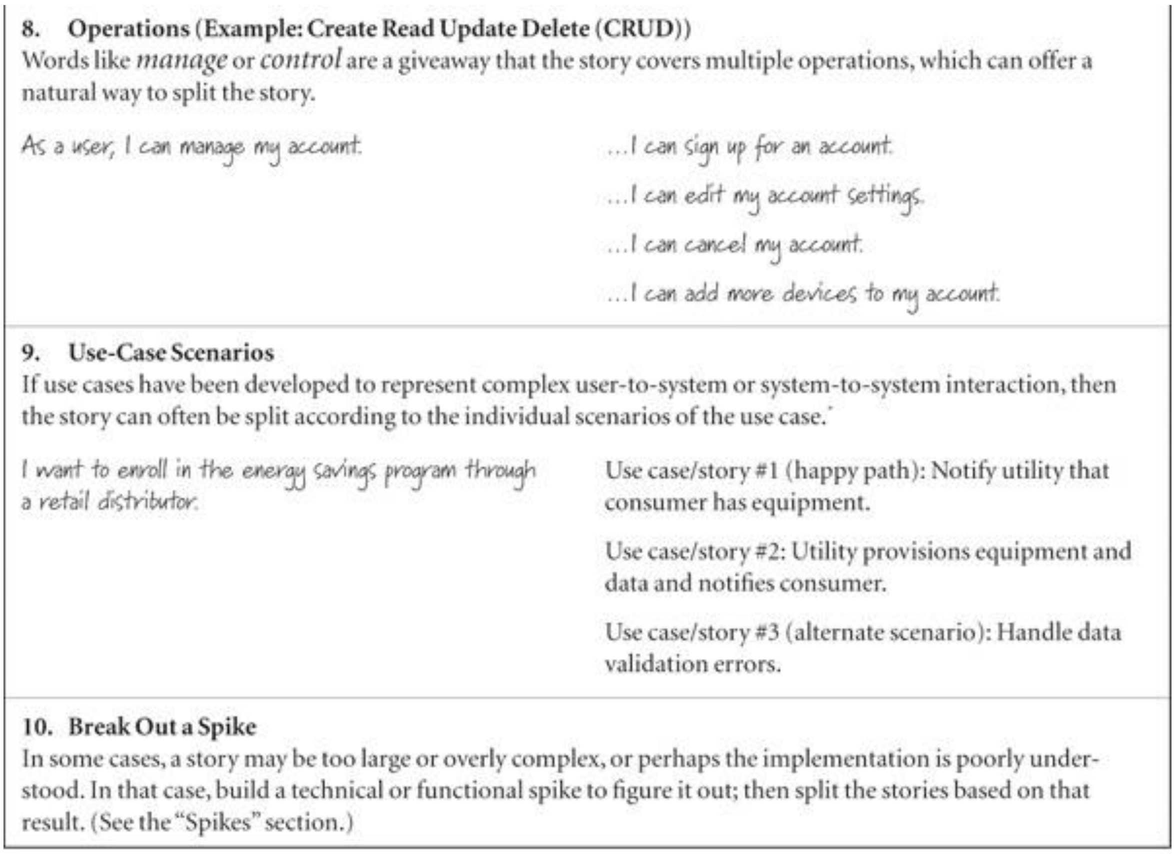
As a utility, I can send messages to customers.	…customers who want their messages:
	…in Spanish
	…in Arabic, and so on.

6. Data Entry Methods
Sometimes complexity is in the user interface rather than the functionality itself. In that case, split the story to build it with the simplest possible UI, and then build the richer UI later.

As a user, I can view my energy consumption in various graphs.	…using bar charts that compare weekly consumption.
	…in a comparison chart, so I can compare my usage to those who have the same or similar household demographics.

7. Defer System Qualities
Sometimes, the initial implementation isn't all that hard, and the major part of the effort is in making it fast or reliable or more precise or more scalable. However, the team can learn a lot from the base implementation, and it should have some value to a user, who wouldn't otherwise be able to do it all. In this case, break the story into successive "ilities."

As a user, I want to see real-time consumption from my meter.	…interpolate data from the last known reading.
	…display real-time data from the meter.



8. Operations (Example: Create Read Update Delete (CRUD)) Words like *manage* or *control* are a giveaway that the story covers multiple operations, which can offer a natural way to split the story.	
As a user, I can manage my account.	…I can sign up for an account. …I can edit my account settings. …I can cancel my account. …I can add more devices to my account.
9. Use-Case Scenarios If use cases have been developed to represent complex user-to-system or system-to-system interaction, then the story can often be split according to the individual scenarios of the use case.*	
I want to enroll in the energy savings program through a retail distributor.	Use case/story #1 (happy path): Notify utility that consumer has equipment. Use case/story #2: Utility provisions equipment and data and notifies consumer. Use case/story #3 (alternate scenario): Handle data validation errors.
10. Break Out a Spike In some cases, a story may be too large or overly complex, or perhaps the implementation is poorly understood. In that case, build a technical or functional spike to figure it out; then split the stories based on that result. (See the "Spikes" section.)	

*The application of use cases in agile development is the entire topic of Chapter 19.

When splitting stories, the team should use an appropriate combination of the previous techniques to consider means of decomposition or multiple patterns in combination. With this skill, the team will be able to move forward at a more rapid pace, splitting user stories at release- and iteration-planning boundaries into bite-size chunks for implementation.

Spikes

Spikes, another invention of XP, are a special type of story used to drive out risk and uncertainty in a user story or other project facet. Spikes may be used for a number of reasons.

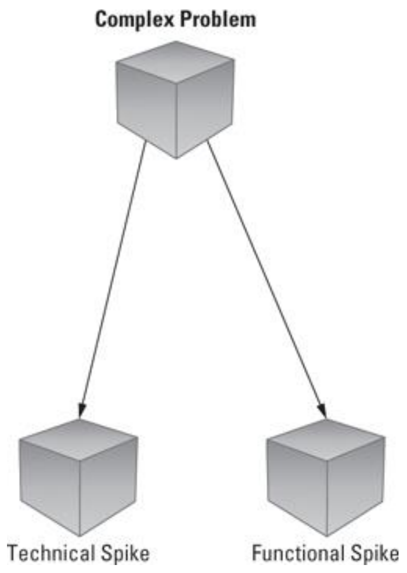
• Spikes may be used for basic research to familiarize the team with a new technology or domain.

• The story may be too big to be estimated appropriately, and the team may use a spike to analyze the implied behavior so they can split the story into estimable pieces.

• The story may contain significant technical risk, and the team may have to do some research or prototyping to gain confidence in a technological approach that will allow them to commit the user story to some future timebox.

• The story may contain significant functional risk, in that although the intent of the story may be understood, it’s not clear how the system needs to interact with the user to achieve the benefit implied.

Technical Spikes and Functional Spikes



Technical spikes are used to research various technical approaches in the solution domain. For example, a technical spike may be used to determine a build-versus-buy decision, to evaluate potential performance or load impact of a new user story, to evaluate specific implementation technologies that can be applied to a solution, or for any reason when the team needs to develop a more confident understanding of a desired approach before committing new functionality to a timebox.

Functional spikes are used whenever there is significant uncertainty as to how a user might interact with the system. Functional spikes are often best evaluated through some level of prototyping, whether it be user interface mock-ups, wireframes, page flows, or whatever techniques are best suited to get feedback from the customer or stakeholders. Some user stories may require both types of spikes. Here’s an example:

As a consumer, I want to see my daily energy use in a histogram so that I can quickly understand my past, current, and projected energy consumption.

In this case, a team might create two spikes:

Technical spike: Research how long it takes to update a customer display to current usage, determining communication requirements, bandwidth, and whether to push or pull the data.

Functional spike: Prototype a histogram in the web portal and get some user feedback on presentation size, style, and charting attributes.

Guidelines for Spikes

Since spikes do not directly deliver user value, they should be used sparingly and with caution. The following are some guidelines for applying user spikes.

Estimable, Demonstrable, and Acceptable

Like other stories, spikes are put in the backlog, estimated, and sized to fit in an iteration. Spike results are different from a story, because they generally produce information, rather than working code. A spike may result in a decision, prototype, storyboard, proof of concept, or some other partial solution to help drive the final results. In any case, the spike should develop just the information sufficient to resolve the uncertainty in being able to identify and size the stories hidden beneath the spike.

The output of a spike is demonstrable, both to the team and to any other stakeholders. This brings visibility to the research and architectural efforts and also helps build collective ownership and shared responsibility for the key decisions that are being taken.

And, like any other story, spikes are accepted by the product owner when the acceptance criteria for the spike have been fulfilled.

The Exception, Not the Rule

Every user story has uncertainty and risk—this is the nature of agile development. The team discovers the right solution through discussion, collaboration, experimentation, and negotiation. Thus, in one sense, every user story contains spike-level activities to flush out the technical and functional risk. The goal of an agile team is to learn how to embrace and effectively address this uncertainty in each iteration. A spike story, on the other hand, should be reserved for the more critical and larger unknowns.

When considering a spike for future work, first consider ways to split the story through the strategies discussed earlier. Use a spike as a last option.

Implement the Spike in a Separate Iteration from the Resulting Stories

Since a spike represents uncertainty in one or more potential stories, planning for both the spike and the resultant stories in the same iteration is risky and should generally be avoided. However, if the spike is small and straightforward and a quick solution is likely to be found, there is nothing wrong with completing the stories in the same iteration. Just be careful.

Story Modeling with Index Cards

Writing and modeling user stories using physical index cards provides a powerful visual and kinesthetic means for engaging the entire team in backlog development. This interactive approach has a number of advantages.

• The physical size of index cards forces a text length limit, requiring the writer to articulate their ideas in just a sentence or two. This helps keep user stories small and focused, which is a key attribute. Also, the tangible and physical nature of the cards gives teams the ability to visually and spatially arrange them in various configurations to help define the backlog.
• Cards may be arranged by feature (or epic) and may be written on the same colored cards as the feature for visual differentiation.
• Cards can also be arranged by size to help developers "see" the size relationships between different stories.
• Cards can be arranged by time or iteration to help evaluate dependencies, understand logical sequencing, see the impact on team velocity, and better align and communicate differing stakeholder priorities.
• The more cards you have, the more work you see, so scoping is a more natural process.

Any team member can write a story card, and the physical act of moving these small, tangible "value objects" around the table creates an interactive learning setting where participants "see and touch" the value they are about to create for their stakeholders.

Experience has shown that teams with a shared vision are more committed to implementing that vision. Modeling value delivery with physical story cards provides a natural engagement model for all team members and stakeholders—one that results in a shared, tangible vision for all to see and experience.

Summary

In this chapter, we provided an overview of the derivation and application of user stories as the primary requirements proxy used by agile teams. Along with background and history, we described the alliteration *card, conversation, and confirmation*, which defines the key elements of a user story. We provided some recommendations for developing good user stories in accordance with the INVEST model and specifically described how *small* stories increase throughput and quality. We also described a set of patterns for splitting large stories into smaller stories so that each resultant story can independently deliver value in an iteration. We also provided guidelines for creating spikes as story-like backlog items for understanding and managing development risk. In conclusion, we suggested that teams apply visual modeling using physical index cards for developing user stories and create a shared vision for implementing user value using this uniquely agile requirements construct.

In the next chapter, we'll strive for a deeper understanding of the users and user personas for whom these user stories are intended.

Chapter 10. Acceptance Testing

What's done, is done.

—Shakespeare, *Macbeth*, Act 3, scene 2

If it isn't tested, it doesn't exist.

—Anonymous agile master

We recently transitioned to agile. But all our testers quit.

—Vignette from Crispin and Gregory [2009]

Why Write About Testing in an Agile Requirements Book?

As a sanity check in preparing for this chapter, I went to my bookshelf and looked at a number of texts on software requirements management, including my own [Leffingwell and Widrig 2003], for guidance on testing whether an application meets its requirements. Of course, I knew I wouldn't find much on testing there, if for no other reason than I knew I hadn't written much. I wasn't surprised that other requirements authors haven't written much of anything on testing either.

So, the question naturally arises: Why do we feel compelled to write about testing now, in a book on agile requirements? The question itself reflects a traditional view, that historically, software requirements were somehow independent of their implementation. They lived a separate life—you could get them (reasonably right) at some point, mostly up front; the developers could actually implement them as intended; they would be tested somewhat independently to assure the system worked as intended; and the customers and users would be happy with the result. Of course, it never really worked that way, but it sure was easier to write about it.

In thinking in lean and agile terms, however, we must take a much more systemic and holistic view. We understand that stories (requirements), implementation (code), and validation (acceptance tests, unit tests, and others) are not separate activities but a continuous refinement of a much deeper understanding; therefore, our thinking is different:

No matter what we thought earlier in the project, *this* functionality is what the user really needs, and it's now implemented, working, and tested in accordance with the continuous discussions and agreements we have forged during development.

Just as importantly, we have instrumented the system (with automated regression tests) such that we can assure this functionality will continue to work as we make future changes and enhancements to the system. Then, and only then, can we declare that our work is complete for this increment.

That is the reason that we have taken a much more systemic view of "requirements" in this book—discussing users, agile teams, agile process, roles, product owners, and whatever else is necessary for a team to develop an application that, in the end, actually solves the user's problem.

However, when describing requirements in book form, the subject is fuzzier and more tangible at the same time. It's *fuzzier* because you can't really tell where a story ends and its acceptance test begins. Are the data elements in a user entry field included in the story, or are they implied requirements attached to the story? Are they really details left for the acceptance test? Or are they perhaps so fine-grained that they may be covered solely in the unit tests for the method that implements it?

And yet, the subject is *more tangible*, because the precise answers to these questions aren't so important. What is important is that we worked through a cycle of incremental information discovery; we collaborated, we negotiated, we refined, we compromised, and we ended up with something that actually works. In addition, we have captured the details of system behavior in a set of tests that will persist for all the time the software continues to provide value to its users. So, the requirements are implemented, complete, and tested.

In this chapter, we'll provide guidance to these questions and an overview of how we achieve quality in our agile requirements practices. In agile, we simply can't do that without a discussion of testing.

Agile Testing Overview

Given that we are taking a systemic view to requirements, across the team, program, and portfolio, we must also take a broader view of agile testing in general, so we'll know the context in which a discussion of acceptance testing can make sense.

Brian Marick, an early XP proponent (and a signer of the Agile Manifesto), has provided much of the thought leadership in this area and has developed a framework that many agilists use to think about testing in an agile paradigm. His philosophy of agile testing is as follows:[1]

Agile testing is a style of testing, one with lessened reliance on documentation, increased acceptance of change, and the notion that a project is an ongoing conversation about quality.

He goes on to describe two main categories of testing: *business-facing* and *technology-facing*tests:[2]

A **business-facing** test is one you could describe to a business expert in terms that would (or should) interest her .... You use words drawn from the business domain: "If you withdraw more money than you have in your account, does the system automatically extend you a loan?"

A **technology-facing** test is one you describe with words drawn from the domain of the programmers: “Different browsers implement JavaScript differently, so we test whether our product works with the most important ones.”

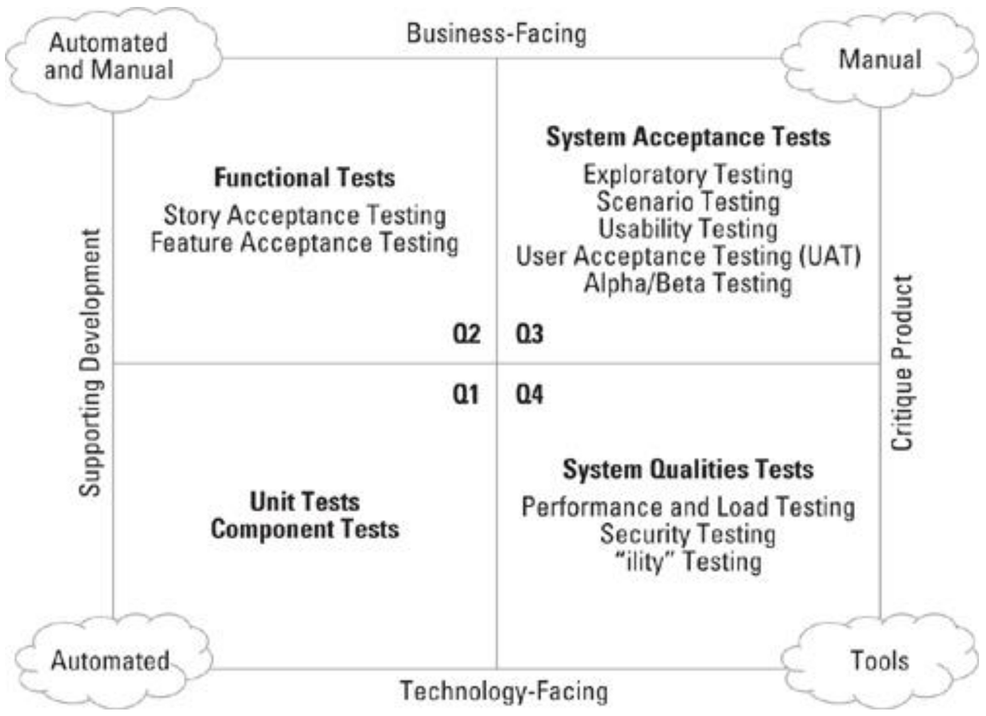
He further categorizes tests, whether business-facing or technology-facing, as being used primarily to either *support programming* or to *critique the product*.

Tests that **support programming** mean that the programmers use them as an integral part ...of programming. For example, some programmers write a test to tell them what code to write next .... Running the test after the [code] change reassures them that they changed what they wanted. Running all the other tests reassures them that they didn’t change behavior they intended to leave alone.

Tests that **critique the product** are not focused on the act of programming. Instead, they look at a finished product with the intent of discovering inadequacies.

In *Agile Testing*, Crispin and Gregory developed these concepts further [2009]. With a few minor adaptations for our context, we find an agile testing matrix in Figure 10-1.

Figure 10-1. The agile testing matrix



In quadrant 1, we find *unit tests* and *component tests*, which are the tests written by developers to test whether the system does what they intended it to do. As indicated on the matrix, these tests are primarily *automated*, because there will be a very large number of them, and they can be implemented in the unit testing environment of choice.

In quadrant 2, we find *functional tests*. In our case, these consist primarily of the story-level acceptance tests that the teams use to validate that each new story works the way the product owner (customer, user) intended. Feature-level acceptance testing is referenced in this quadrant as well. Many of these tests can be automated, as we'll see later, but some of these tests are likely to be manual.

In quadrant 3, we find *system acceptance tests*, which are system-level tests to determine whether the aggregate behavior of the system meets its usability and functionality requirements, including the many variations (scenarios) that may be encountered in actual use. These tests are largely manual in nature, because they involve users and testers using the system in actual or simulated deployment and usage scenarios.

In quadrant 4, we find *system qualities tests*, which are used to determine whether the system meets its nonfunctional requirements. Such tests are typically supported by a class of testing tools, such as load and performance testing tools, which are designed specifically for this purpose.

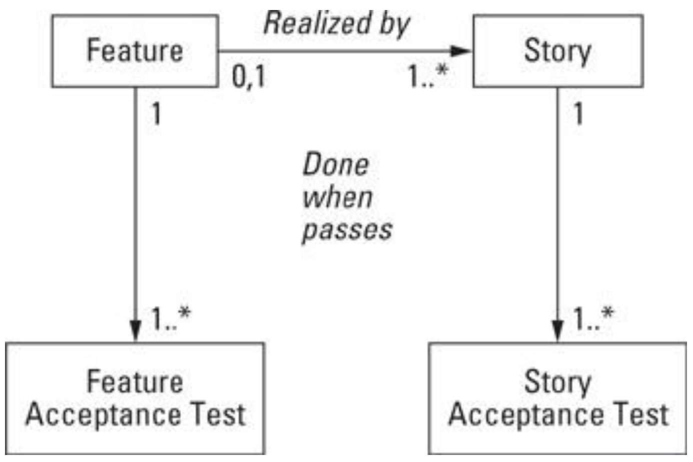
With this perspective, we have a way to think about the different types of testing we'll need to do to assure that a system performs as expected.

What Is Acceptance Testing?

The language around testing is as overloaded as any other domain in software development, so the words *acceptance testing* mean different things to different people. Indeed, there are two different uses of the term in the agile testing matrix. In Figure 10-1, quadrant 2, we see*functional* and *story and feature acceptance tests*, and in quadrant 3, we see *system-level acceptance tests*. In this chapter, we'll focus on two of these quadrants, Q2 and Q1. InChapter 17, Nonfunctional Requirements, we'll revisit the testing process to look at testing practices that support the tests indicated in quadrants 3 and 4.

In quadrant 2, we find both feature and story acceptance tests, which are used to assure that features and stories, respectively, are *done*, as we illustrate in our model in Figure 10-2.

Figure 10-2. Features and stories cannot be considered done until they pass one or more acceptance tests.



Story Acceptance Tests

The majority of the testing work done by agile teams is in the development, execution, and regression testing of story acceptance tests (SATs), so we will focus on those first. Story acceptance tests are functional tests intended to assure that the implementation of each new user story (or other story type) delivers the intended behavior. If all the new stories work as intended, then each new increment of software is delivering value and provides assurances that the project is progressing in a way that will ultimately satisfy the needs of the users and business owners. Generally, the following is true.

• They are written in the language of the business domain (they are business-facing tests from quadrant 2).

• They are developed in a conversation between the developers, testers, and product owner.

• Although anyone can write tests, the product owner, as business owner/customer proxy, is the primary owner of the tests.

• They are black-box tests in that they verify only that the outputs of the system meet the conditions of satisfaction, without concern for how the result is achieved.

• They are implemented during the course of the iteration in which the story itself is implemented.

This means that new acceptance tests are developed for every new story. If a story does not pass its test, the teams get no credit for the story, and the story is carried over into the next iteration, where the code or the test, or both, are reworked until the test passes.

Characteristics of Good Story Acceptance Tests

If stories are the workhorse of agile development—the key proxy artifact that carries the value stream to the customer—then story acceptance tests are the workhorse of agile testing, so teams spend much time defining, refining, and negotiating the details of these tests. That's because, in the end, *it is the details of these tests that define the final, agreed-to behavior of the system*. Therefore, writing good story acceptance tests is a prime factor in delivering a quality system. Good story acceptance tests exhibit the characteristics described in the following sections.

They Test Good User Stories

An attribute of a good SAT is that it is associated with a good user story. In Chapter 6, we described the INVEST model for user stories and the quality of our acceptance tests are fully dependent on the native quality of the story itself. In particular, the user story to which a test is associated has to be independent, small, and testable. If the development of an acceptance test illustrates that the story is otherwise, then the story itself must be refactored until it meets these criteria. To get a good acceptance test, we may need to refactor the story first. For example, the following:

As a consumer, I am always aware of my current energy costs.

becomes this:

As a consumer, I always see current energy pricing reflected on my portal and on-premise devices so that I know that my energy usage costs are accurate and reflect any utility pricing changes.

They Are Relatively Unambiguous and Test All the Scenarios

Since the story itself is a lightweight (even potentially throwaway) expression of intent, the acceptance test carries the detailed requirements for the story, now and into the future. As such, there can be little ambiguity about the details in the story acceptance test. In addition, the acceptance test must test all the scenarios implied by the story. Otherwise, the team won't know when the story is sufficiently complete in order to be able to be presented to the product owner for acceptance. Here's an example:

Story:

As a consumer, I always see current energy pricing reflected on my portal and on-premise devices so that I know that my energy usage costs are accurate and reflect any utility pricing changes.

Acceptance test:

1. *Verify the current pricing is always used and the calculated numbers are displayed correctly on the portal and each on-premise device (see attachment for formats).*
2. *Verify the pricing and the calculated numbers are updated correctly when the price changes.*
3. *Verify the "current price" field itself is updated according to the scheduled time.*
4. *Verify the info/error messages when there is a fault in the pricing (see approved error messages attached).*

They Persist

One of the mysteries about agile, and indeed a key impediment to adoption, is a commonsense question: “If developers don’t document much and there are no software requirements specifications as such, how are we supposed to keep track of what the system actually does? After all, we are the ones responsible for assuring that it actually works, now and in the future. Isn’t that something we have to know, not just once, but in perpetuity?”

The answer is yes, indeed, we do have to know how it works, and we have to routinely regression test it to make sure it continues to work. We do that primarily by persisting and automating (wherever possible) acceptance tests and unit tests (discussed later).

User stories can be safely thrown away after implementation. That keeps them lightweight, keeps them team friendly, and fosters negotiation, but acceptance tests persist for the life of the application. We have to know that the current price field didn’t just get updated once when we tested it but that it gets updated every time the price changes, even when the application itself has been modified.

Acceptance Test-Driven Development

Beck [2003] and others have defined a set of XP practices for agility described under the umbrella label of test-driven development (TDD). In TDD, the focus is on writing the unit test before writing the code. For many, TDD is an assumed part of agile development and is straightforward in principle.

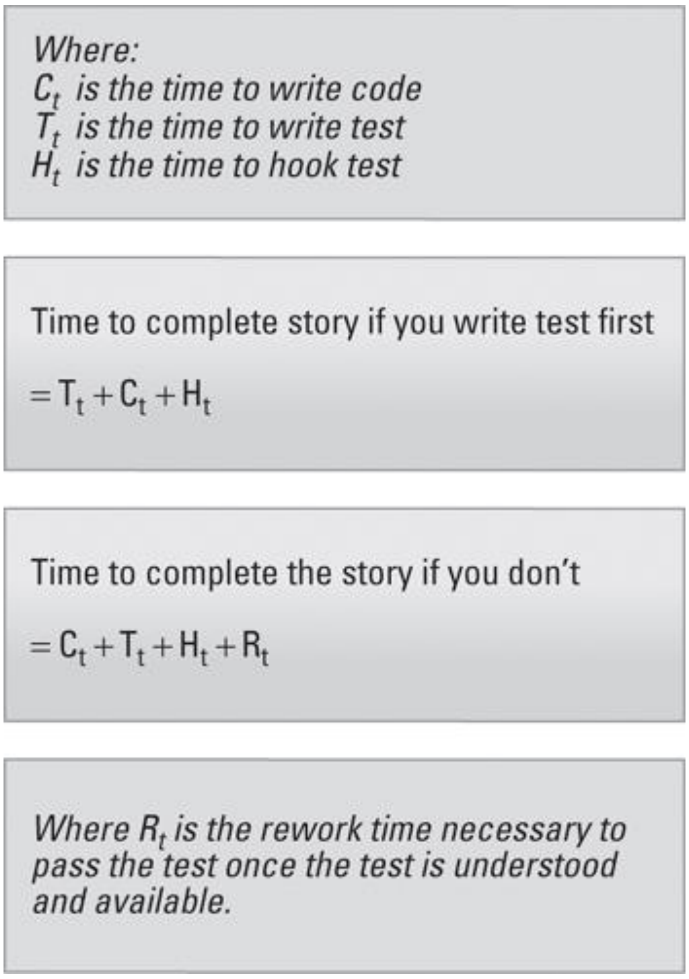
1. Write the test first. Writing the test first forces the developer to understand the required behavior of the new code.
2. Run the test, and watch it fail. Because there is as yet no code to be tested, this may seem silly initially, but this accomplishes two useful objectives: it tests the test itself and any test harnesses that hold the test in place, and it illustrates how the system will fail if the code is incorrect.
3. Write the minimum amount of code that is necessary to pass the test. If the test fails, rework the code or the test as necessary until a module is created that routinely passes the test.

In XP, this practice was primarily designed to operate in the context of unit tests, which are developer-written tests (also code) that test the classes and methods that are used. These are a form of “white-box testing” because they test the internals of the system and the various code paths that may be executed.

However, the philosophy of TDD applies equally well to story acceptance testing as it does to unit testing. This is called *acceptance test-driven development*, and whether it is adopted formally or informally, many teams write the story acceptance test first, before developing the code. The acceptance tests serve to record the decisions made in the conversation (card, conversation, confirmation) so that the team understands the specifics of the behavior the card represents. The code follows logically thereafter.

Proponents argue (correctly, we believe) that writing the acceptance test first is lean thinking that reduces waste and substantially increases the productivity of the team. This was illustrated best in the simple equation that Amir Kolsky, of NetObjectives, showed on a whiteboard. As shown in Figure 10-3, Amir wrote this:[3]

Figure 10-3. The simple math behind acceptance test-driven development



Where:
C_t is the time to write code
T_t is the time to write test
H_t is the time to hook test

Time to complete story if you write test first

$= T_t + C_t + H_t$

Time to complete the story if you don't

$= C_t + T_t + H_t + R_t$

Where R_t is the rework time necessary to pass the test once the test is understood and available.

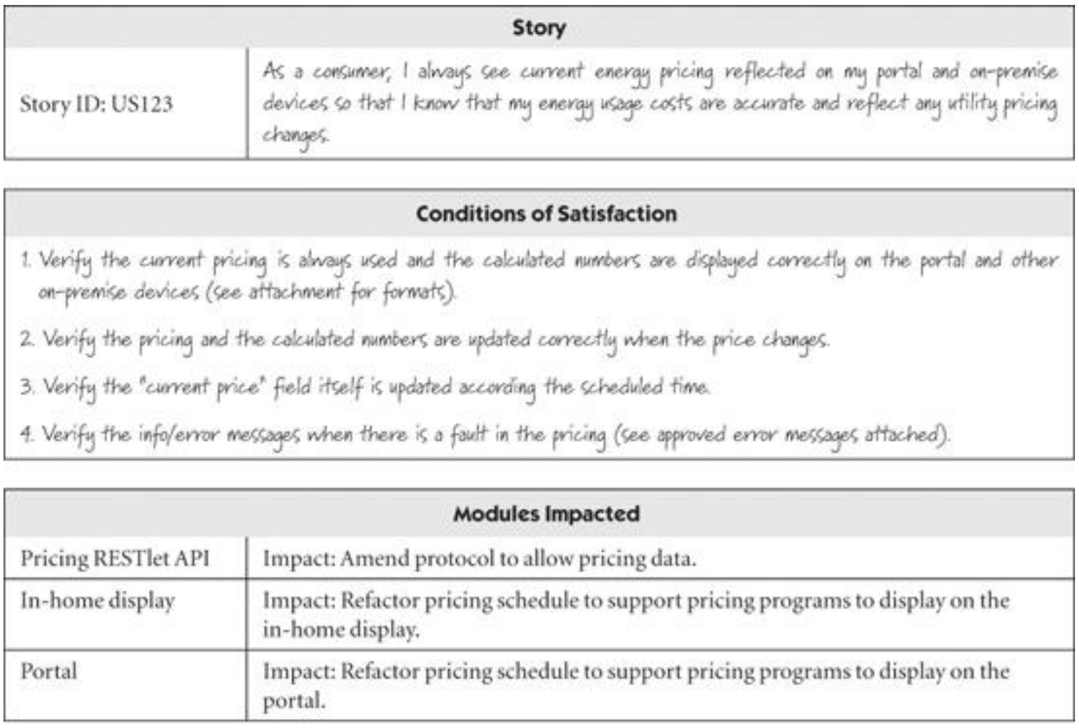
If R_t is 0, of course, then there is no savings. However, we all know that R_t isn't always zero, and since finalizing the test finalizes our understanding of the required behavior, why not write the test first, just to be sure?

Acceptance Test Template

At each iteration boundary or whenever a story is to be implemented, it comes as no surprise to the team that they need to create an acceptance test that further refines the details of a new story and defines the conditions of satisfaction that will tell the team when the story is ready for acceptance by the product owner. In addition, in the context of a team and a current iteration, the domain of the story is pretty well-established, and certain patterns of activities result, which can guide the team to the work necessary to get the story accepted into the baseline.

To assist in this process, it can be convenient to the team to have a checklist—a simple list of things to consider—to fill out, review, and discuss each time a new story appears. Crispin and Gregory [2009] provide an example of such a story acceptance testing checklist in their book. Based on our experience using this checklist, we provide an example from the case study inTable 10-1.

Table 10-1. An Acceptance Testing Example from the Case Study



Story	
Story ID: US123	As a consumer, I always see current energy pricing reflected on my portal and on-premise devices so that I know that my energy usage costs are accurate and reflect any utility pricing changes.

Conditions of Satisfaction
1. Verify the current pricing is always used and the calculated numbers are displayed correctly on the portal and other on-premise devices (see attachment for formats).
2. Verify the pricing and the calculated numbers are updated correctly when the price changes.
3. Verify the "current price" field itself is updated according the scheduled time.
4. Verify the info/error messages when there is a fault in the pricing (see approved error messages attached).

Modules Impacted	
Pricing RESTlet API	Impact: Amend protocol to allow pricing data.
In-home display	Impact: Refactor pricing schedule to support pricing programs to display on the in-home display.
Portal	Impact: Refactor pricing schedule to support pricing programs to display on the portal.

Documents Impacted	
User guide	Impact: Add new section on pricing.
Online help	Impact: Update online help to reflect pricing programs.
Release notes	Impact: Document defects in release notes.
Utility guide	Impact: Document pricing schedule changes.

Test Case Outline	
Test ID: ☒ **Manual** ☐ Automatic	Outline: 1. Check pricing: When there is no pricing info for a user: 2. Change of pricing: • When there is a pricing change in all allowed ways. • Effective in the future. • Effective in the past before the current pricing. • Effective in the past but later than the current pricing. 3. Our current release does not support pricing change in the middle of a billing cycle. 4. Check the dashboard billing period consumption and the current bill to date.

Communications		
Internal	Involved parties: marketing, sales, product management.	Message: This is a new marketable feature.
External	Involved parties: utilities.	Message: This is a new feature to support new programs.

Since each team is in a different context, their templates will differ, but the simple act of creating a template as a reminder of all the things to think about benefits the team and increases the velocity with which they can further elaborate and acceptance test a new story.

Automated Acceptance Testing

Because acceptance tests run at a level above the code, there are a variety of approaches to executing these tests, including manual tests. However, manual tests pile up very quickly (the faster you go, the faster they grow), and eventually, the number of manual tests required to run a regression slows down the team and introduces delays in the value stream.

To avoid this problem, most teams know that they have to automate most of their acceptance tests. They use a variety of tools to do so, including database-driven tests, Web UI testing tools, and automated tools for record and playback.[4] However, many agile teams have discovered that some of these methods are labor-intensive and can be somewhat brittle and difficult to maintain because they often couple so tightly to the specific implementation.

A better approach is to take a higher level of abstraction that works directly against the business logic of the application and one that is not encumbered by the presentation layer or other implementation details.

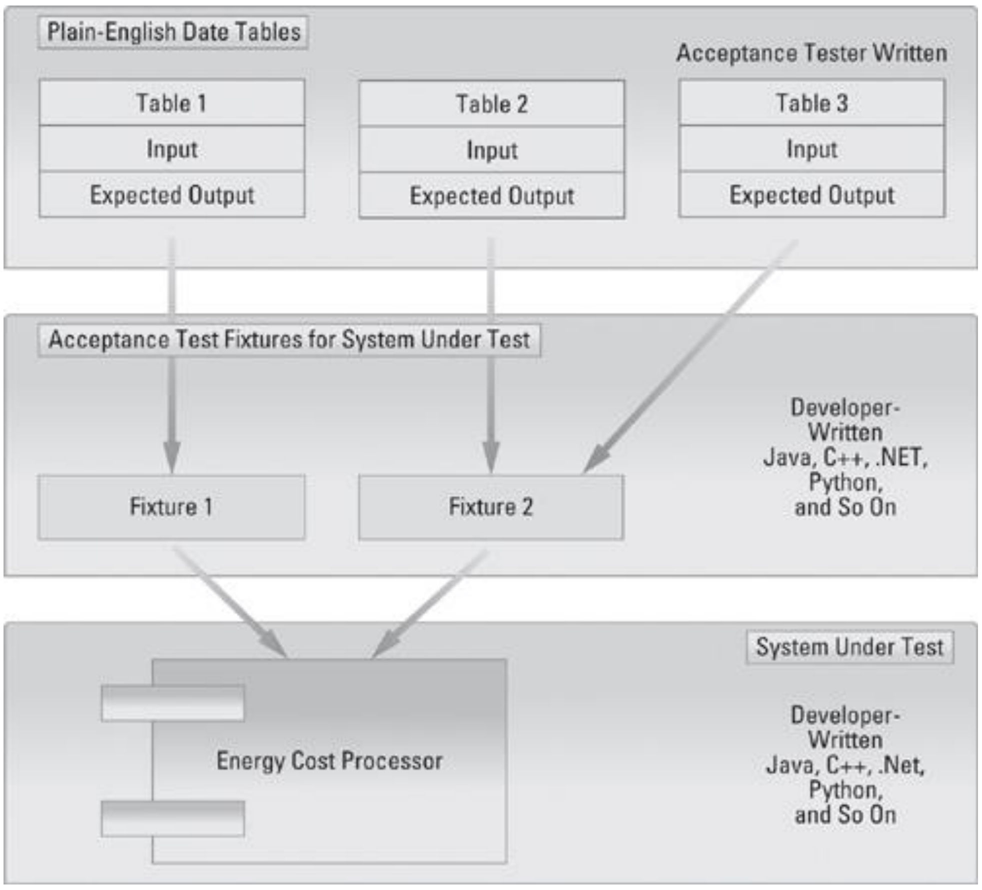
Automated Acceptance Testing Example: The FIT Approach

One such method is the Framework for Integrated Tests (FIT) method created by Ward Cunningham [Mugridge and Cunningham 2005]. This open source framework was designed to help with the automation of acceptance testing in a fast-moving agile context.

The FIT approach mirrors the unit testing approach in that the tests are created and run against the system under test, but they are not part of the system itself. FIT is a scriptable framework that supports tests being written in table form (any text tool will work) and saved for input as HTML. Therefore, these tests can be constructed in the business language (input and expected results) and can be written by developers, product owners, testers, or anyone on the team capable of building the necessary scripts.

Another open source component, FitNesse, is a wiki/web-based front end for creating text tables for FIT that also provides some test management capability. FIT uses data-driven tables for individual tests, coupled with fixtures or methods written by the developers to drive the system under test, as Figure 10-4 illustrates.

Figure 10-4. Acceptance test framework for the costing processor


Plain-English Date Tables
Acceptance Tester Written
Table 1
Input
Expected Output
Table 2
Input
Expected Output
Table 3
Input
Expected Output
Acceptance Test Fixtures for System Under Test
Developer-Written Java, C++, .NET, Python, and So On
Fixture 1
Fixture 2
System Under Test
Energy Cost Processor
Developer-Written Java, C++, .Net, Python, and So On

During the course of each iteration, new acceptance tests are developed and validated for each new story in the iteration, and these tests are then added to the regression test suite. These suites of acceptance tests can be run automatically against the system under test at any time to assure that the build is not broken by the new code.

As with unit tests, the FIT approach requires continuous involvement by the developers in establishing the acceptance test strategy and then writing the fixtures to support the test, illustrating yet again that the define/build/test team is the necessary structure for effective agile development. As always, the team's goal is to develop and automate tests within the course of the iteration in which the new functionality is introduced.

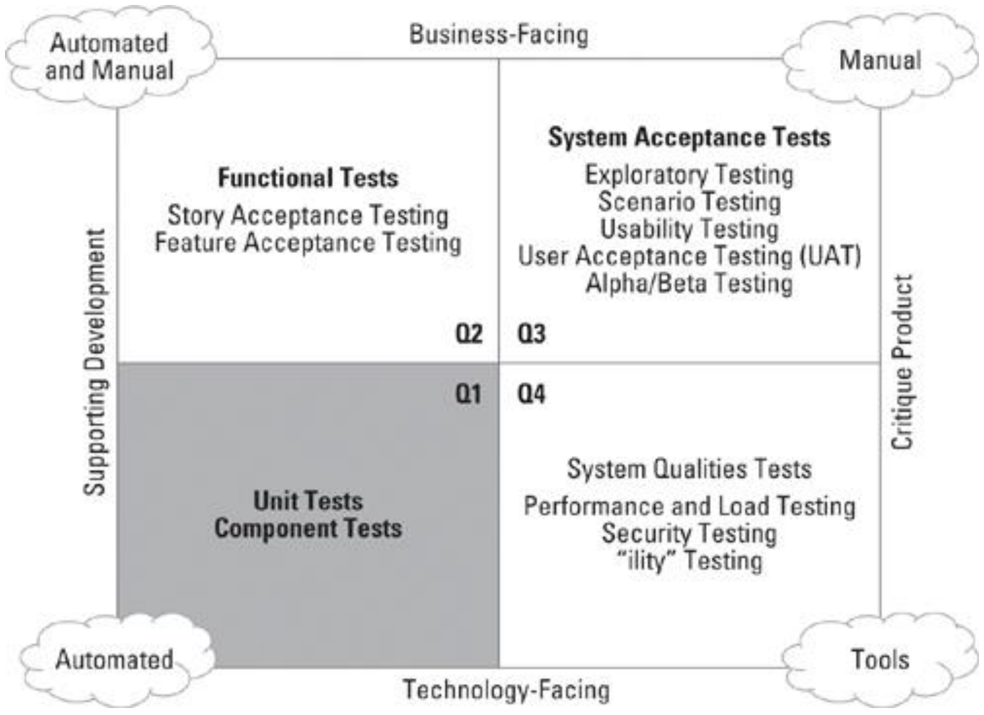
In some application domains, FIT does not provide an appropriately configurable framework. Sometimes, teams must build custom frameworks that mirror this approach but work in the technologies of their implementation.

In any case, whatever can't be automated eventually slows the team down, so continuous investments in testing automation infrastructure are routine items on a team's backlog, and like any other backlog item, they must be appropriately prioritized by the product owner to achieve sustainable high velocity.

Unit and Component Testing

Before we leave this chapter, we must drill down one more level into testing and discuss quadrant 1 in the agile testing picture. In quadrant 1, we see *unit tests* and *component tests*, which are technology-facing tests as they are implemented and executed by the development team. Together with feature and story acceptance tests, they also complete the *support development* side of the agile testing picture, as Figure 10-5 illustrates.

Figure 10-5. Unit testing in quadrant 1



Unit Testing

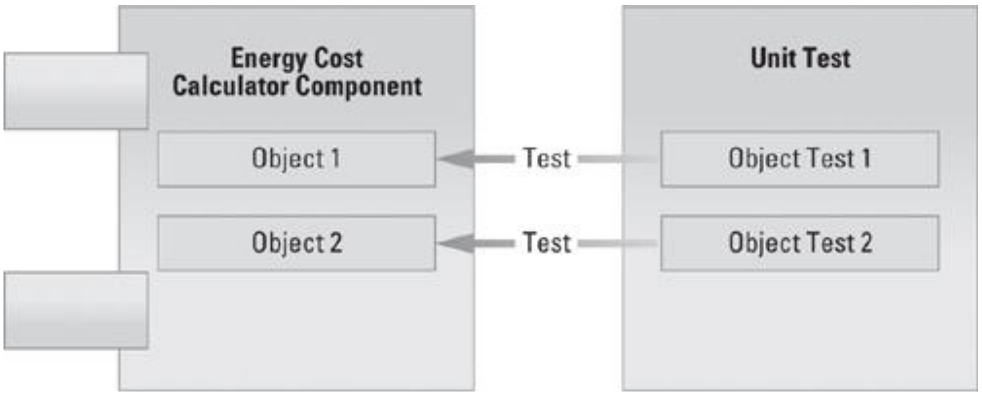
Unit testing is the white-box testing whereby developers write test code to test the production code they developed for the system. In so doing, the understanding of the user story is further refined, and additional details about a user story can be found in the unit tests that accompany the code. For example, the presentation syntax and range of legal values for*current price field* can likely be found in the unit tests, rather than the acceptance tests, because otherwise the acceptance tests are long, are unwieldy, and cause attention to the wrong level of detail.

A comprehensive unit test strategy prevents QA and test personnel from spending most of their time finding and reporting on code-level bugs and allows the team to move its focus to more system-level testing challenges. Indeed, for many agile teams, the addition of a comprehensive unit test strategy is a key pivot point in their move toward true agility—and one that delivers the “best bang for the buck” in determining overall system quality.

The history of agile unit testing closely follows the development of XP, because XP's test-first practices drove developers to create low-level tests for their code prior to, or concurrent with, the development of the code itself. The open source community has built unit testing frameworks to cover most forms of testing, including Java, C, C++, XML, HTTP, and Python, so there are unit testing frameworks for most languages and coding constructs an agile developer is likely to encounter.

These frameworks provide a harness for the development and maintenance of unit tests and for automatically executing unit tests against the system under development. Unit testing the energy cost calculator might be a matter of writing unit tests against every object in the component, as Figure 10-6 illustrates.

Figure 10-6. Unit testing the energy cost calculator



The unit tests themselves are not part of the system under test and therefore do not affect the performance of the system at runtime.

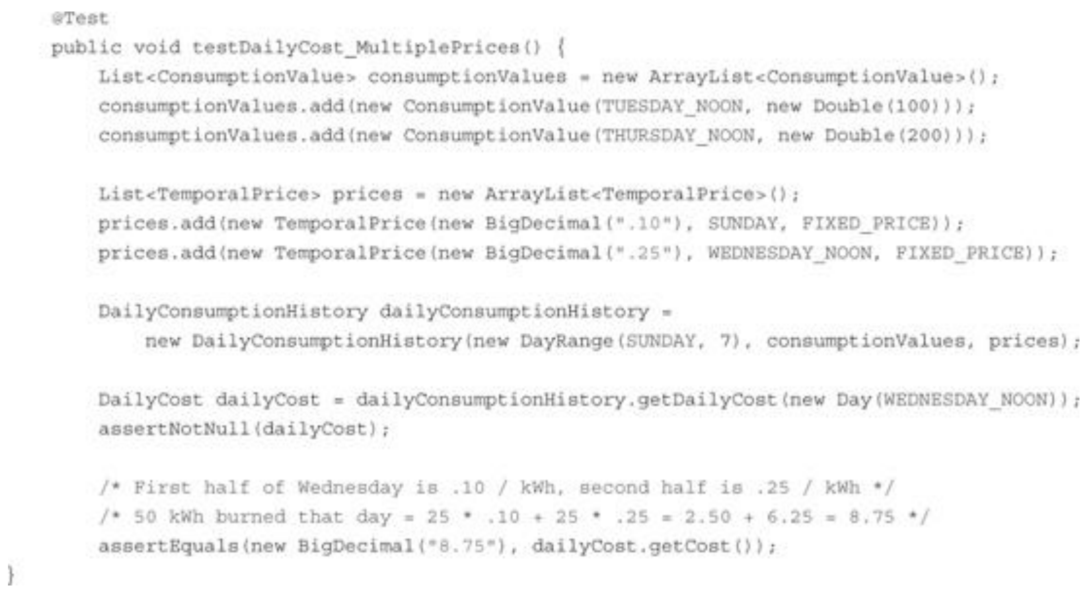
Unit Testing in the Course of the Iteration

Because the unit tests are written before or concurrently with the code and because the unit testing frameworks include test execution automation, all unit testing can be accomplished within the iteration. Moreover, the unit test frameworks hold and manage the accumulated unit tests, so regression testing automation for unit tests is largely free for the team. Unit testing is a cornerstone practice of software agility, and any investments a team makes toward more comprehensive unit testing will be well rewarded in quality and productivity.

Figure 10-7 shows an example of a unit test for Tendril's energy costing module that takes the consumer's time-based pricing structure and runs a small sample that will exercise many of the costing module's pathways. This test ensures that the costing algorithm effectively accommodates price changes that may occur at times other than standard day boundaries. Note that many more unit tests will be required before the costing module can be considered*done*. This is just one of many unit tests using the JUnit testing platform. This unit test example is one of many associated with calculating current costing, but it is automated and is an integral part of the regression testing package. The sample shows another useful feature in that there are comments embedded in the test to explain what the test results are supposed to be.

Figure 10-7. A sample unit test from the case study

Thanks to Ben Hoyt of Tendril for this example.



```
    @Test
    public void testDailyCost_MultiplePrices() {
        List<ConsumptionValue> consumptionValues = new ArrayList<ConsumptionValue>();
        consumptionValues.add(new ConsumptionValue(TUESDAY_NOON, new Double(100)));
        consumptionValues.add(new ConsumptionValue(THURSDAY_NOON, new Double(200)));

        List<TemporalPrice> prices = new ArrayList<TemporalPrice>();
        prices.add(new TemporalPrice(new BigDecimal(".10"), SUNDAY, FIXED_PRICE));
        prices.add(new TemporalPrice(new BigDecimal(".25"), WEDNESDAY_NOON, FIXED_PRICE));

        DailyConsumptionHistory dailyConsumptionHistory =
            new DailyConsumptionHistory(new DayRange(SUNDAY, 7), consumptionValues, prices);

        DailyCost dailyCost = dailyConsumptionHistory.getDailyCost(new Day(WEDNESDAY_NOON));
        assertNotNull(dailyCost);

        /* First half of Wednesday is .10 / kWh, second half is .25 / kWh */
        /* 50 kWh burned that day = 25 * .10 + 25 * .25 = 2.50 + 6.25 = 8.75 */
        assertEquals(new BigDecimal("8.75"), dailyCost.getCost());
}
```

Component Testing

In a like manner, component testing is used to test larger-scale components of the system. Many of these are present in various architectural layers, where they provide services needed by features or other components.

Testing tools and practices for implementing component tests vary according to the nature of the component. For example, unit testing frameworks can hold arbitrarily complex tests written in the framework language (Java, C, and so on), so many teams use their unit testing frameworks to build component tests. They may not even think of them differently.

Acceptance testing frameworks, especially those at the level of *http Unit* and *XML Unit*, are also employed. In other cases, developers may use testing tools or write fully custom tests in any language or environment that is most productive for them.

Summary

In this chapter, we described acceptance testing as an integral part of agile requirements management. If a requirement (feature or story) is not tested, unless it's simply work in process, which we indeed try to minimize, it doesn't really have any value to the team or to the user. With this discussion, we described the agile testing approach to assure that each new story works as intended, as it is implemented. This covers quadrant 2, functional testing of the agile testing matrix.

We also described the testing necessary to assure systematic quality from the perspective of quadrant 1, unit and component tests. Unit tests are the lowest level of tests written by the developer to assure that the actual code (methods, classes, and functions) works as intended. Component tests are higher-level tests that are written by the team to assure that the larger components of the system, which aggregate functionality along architectural boundaries, also work as intended. We also described how the team must endeavor to automate all the testing that is possible or else they will simply build a pile of manual regression tests that will eventually decrease velocity and slow down value delivery.

In Chapter 17, Nonfunctional Requirements, we'll describe acceptance testing practices associated with quadrant 4—system qualities tests.

Chapter 18. Requirements Analysis Toolkit

There is no sense in being precise about something, when you don't even know what you are talking about.

—John Von Neumann

Smaller projects, such as those that can operate solely at the Team level, are usually so well contained that communication among the stakeholders is not a big issue. The team usually has easy access to each other, and it's easy to resolve ambiguities and confusion. Just lean over to the next workstation, and ask the product owner what they meant with a story!

However, as projects grow in scope and size, the communication pathways become more and more complex, and opportunities for misunderstandings arise. Indeed, as projects grow to the Program level, it's pretty much a given that different means of communication become necessary. In addition, there are cases in which the ambiguity of imperfect requirements communication is simply not tolerable, particularly when the stories deal with life-and-death issues or when the erroneous behavior of a system could have extreme financial or legal consequences.

Communication has always been an imperfect vehicle. Misunderstandings abound, and it's common to say or hear, "This story is perfectly clear. Why don't you understand it?" Indeed, it may be clear to the story writer, but others may not find it so obvious at all. For example, think back to the "estimating" exercise in Chapter 8 for an "obvious" statement of requirements, such as "count the pages in the workbook," and look at all the confusions that arise within the scope of that simple exercise. Now, consider your real world.

Even with user stories, so helpful in bridging the gap between developer and user communities, our pidgin language may be able to communicate fairly simple things in a simple way, sufficient to buy some beads or build a software widget, but sometimes far more precision is required (How exactly is that cardiac pacemaker algorithm supposed to work...?).

In other words, when we are building complex systems, there are clearly times when we need alternate and far more precise communication mechanisms.

If the description of the story is too complex for natural language and if the business cannot afford to have the specification misunderstood, the team should augment the story with a more precise specification method.

In this chapter, we'll describe a number of *requirements analysis techniques*, more technical methods for specifying system behavior that the team can use to resolve ambiguity and build more assuredly safe and reliable systems. Such methods include the following:

- Activity diagrams (flowcharts)
- Sample reports
- Pseudocode
- Decision tables and decision trees
- Finite state machines
- Message sequence diagrams
- Entity-relationship diagrams
- Use cases

And there are many others.

We won't attempt to describe any of these techniques in detail since each is worthy of a chapter of its own. But we can provide a brief introduction to each so that you'll have a sense of what to use and when to use it.

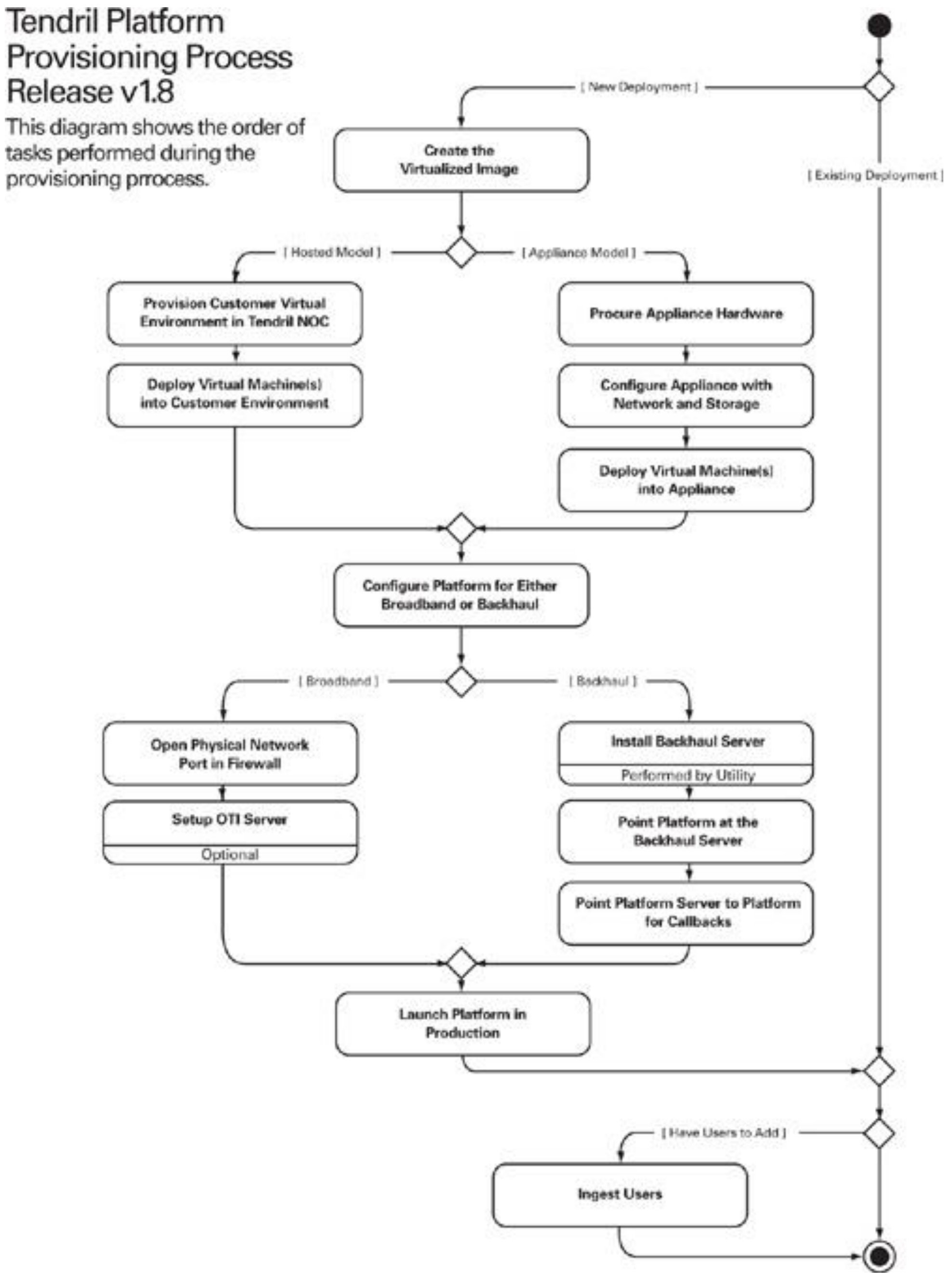
Where possible, only one or two of these technical methods should be used to augment natural-language stories for a system. This simplifies the nontechnical reviewer's task of reading and understanding these special elements. If all the systems developed by an organization fall into one application domain, such as telephone switching systems, perhaps the same technical method can be used for all the systems. But in most organizations, it's unrealistic to mandate a single technique for all stories in all systems. Story writers and analysts need to select an approach that best suits the individual situation and help the users and reviewers understand how the technique expresses system behavior.

In this chapter, we'll describe each of these methods in summary form.

Activity Diagrams

Flowcharts and their new and more precise incarnation, the UML activity diagram, have the advantage of reasonable familiarity. Even people with no computer-related training or background generally know what a flowchart is. Teams apply activity diagrams, such as the Tendril one in Figure 18-1, that specify the major steps and process steps necessary to provision a new release of a product. They could have written a long and involved text-based procedure, or they could simplify the steps into a single understandable flow diagram as inFigure 18-1. Although the same information could have been presented in a number of other forms, the UML notation provides a visual representation that is fairly easy to understand and relatively unambiguous.

Figure 18-1. Tendril provisioning diagram in activity diagram format

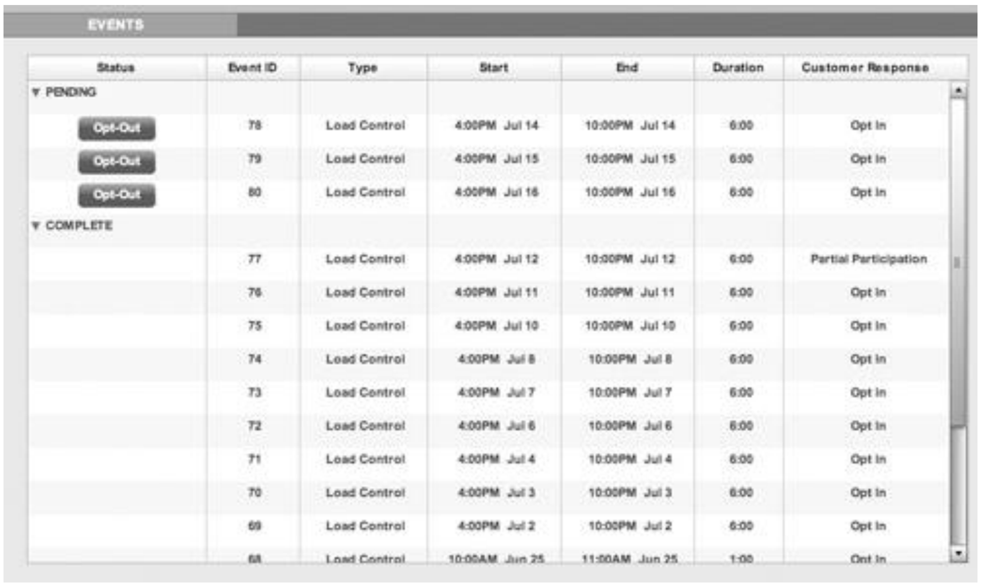

Tendril Platform Provisioning Process Release v1.8
This diagram shows the order of tasks performed during the provisioning process.
[New Deployment]
[Existing Deployment]
Create the Virtualized Image
[Hosted Model]
[Appliance Model]
Provision Customer Virtual Environment in Tendril NOC
Deploy Virtual Machine(s) into Customer Environment
Procure Appliance Hardware
Configure Appliance with Network and Storage
Deploy Virtual Machine(s) into Appliance
Configure Platform for Either Broadband or Backhaul
[Broadband]
[Backhaul]
Open Physical Network Port in Firewall
Setup OTI Server
Optional
Install Backhaul Server
Performed by Utility
Point Platform at the Backhaul Server
Point Platform Server to Platform for Callbacks
Launch Platform in Production
[Have Users to Add]
Ingest Users

Sample Reports

Like the UI challenge we described in Chapter 7, users often don't know what they want to see in a report or other output until they see the report. Many times, teams churn code through a series of iterations, trying to get the right data presented in the right way, without simply stepping back and asking themselves, "What is it that they really want to see, and how do they want to see it?"

In these cases, a sample report format with mock data, generated by any number of desktop tools, might be all that is necessary to resolve most of the ambiguity. If the system is algorithmically intensive and the user cannot evaluate the report format without some real data and results, the team may need to invest some time in producing some real data. Figure 18-2 shows an example of a mock-up report.

Figure 18-2. Sample event report format from case study



Status	Event ID	Type	Start	End	Duration	Customer Response
▼ PENDING						
Opt-Out	78	Load Control	4:00PM Jul 14	10:00PM Jul 14	6:00	Opt In
Opt-Out	79	Load Control	4:00PM Jul 15	10:00PM Jul 15	6:00	Opt In
Opt-Out	80	Load Control	4:00PM Jul 16	10:00PM Jul 16	6:00	Opt In
▼ COMPLETE						
	77	Load Control	4:00PM Jul 12	10:00PM Jul 12	6:00	Partial Participation
	76	Load Control	4:00PM Jul 11	10:00PM Jul 11	6:00	Opt In
	75	Load Control	4:00PM Jul 10	10:00PM Jul 10	6:00	Opt In
	74	Load Control	4:00PM Jul 8	10:00PM Jul 8	6:00	Opt In
	73	Load Control	4:00PM Jul 7	10:00PM Jul 7	6:00	Opt In
	72	Load Control	4:00PM Jul 6	10:00PM Jul 6	6:00	Opt In
	71	Load Control	4:00PM Jul 4	10:00PM Jul 4	6:00	Opt In
	70	Load Control	4:00PM Jul 3	10:00PM Jul 3	6:00	Opt In
	69	Load Control	4:00PM Jul 2	10:00PM Jul 2	6:00	Opt In
	68	Load Control	10:00AM Jun 25	11:00AM Jun 25	1:00	Opt In

Spikes are frequently invoked as research items to develop and validate sample reports with the product owner and users.

Pseudocode

As the term implies, *pseudocode* is a "quasi" programming language, which combines the informality of natural language with the strict syntax and control structures of aprogramming language. In the extreme form, pseudocode consists of combinations of the following:

• Imperative sentences with a single verb and a single object

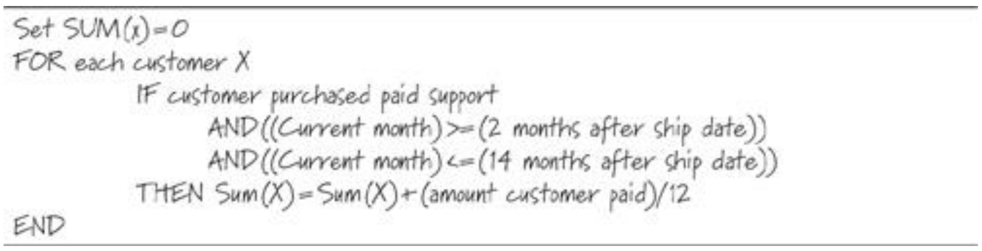
• A limited set, typically not more than 40 to 50, of "action-oriented" verbs from which the sentences must be constructed

• Decisions represented with a formal IF-ELSE-ENDIF structure

• Iterative activities represented with DO-WHILE or FOR-NEXT structures

Figure 18-3 shows an example of a pseudocode specification of an algorithm for calculating deferred-service revenue earned within a given month in a business application. The text of the pseudocode is indented in an outline-style format in order to show "blocks" of logic. The combination of the syntax restrictions and the format and layout of the text greatly reduces the ambiguity of what could otherwise be a very difficult and error-prone story. (It certainly was before we wrote this pseudocode.) At the same time, it should be possible for a nonprogramming person to read and understand the story's function in the form shown inFigure 18-3.

Figure 18-3. Example of pseudocode



```
Set SUM(x)=0
FOR each customer X
        IF customer purchased paid support
              AND((Current month)>=(2 months after ship date))
              AND((Current month)<=(14 months after ship date))
        THEN Sum(X)=Sum(X)+(amount customer paid)/12
END
```

Decision Tables and Decision Trees

It's common to see a story that deals with a combination of inputs; different combinations of those inputs lead to different actions or outputs. Suppose, for example, that we have a system with three inputs—A, B, and C—and we see a story that starts with a pseudocode-like statement: "If A is true, then if B is also true, do action X, unless C is true, in which case the required action is Y." The combination of IF-THEN-ELSE clauses quickly becomes tangled, especially if, as in this example, it involves nested IFs. Typically, nontechnical users are not sure that they understand any of it, and nobody is sure whether all the possible combinations and permutations of A, B, and C have been covered.

The solution in this case is to enumerate all the combinations of inputs and to describe each one explicitly in a decision table. In our example, if the only permissible values of the inputs are "true" and "false," we have 2^3, or eight, combinations. These can be represented in a table containing eight columns. We would then list the actions for each of those eight combinations. Figure 18-4 illustrates a real-life problem that many users encounter, a printer malfunction.

Figure 18-4. Decision table for debugging a printer failure

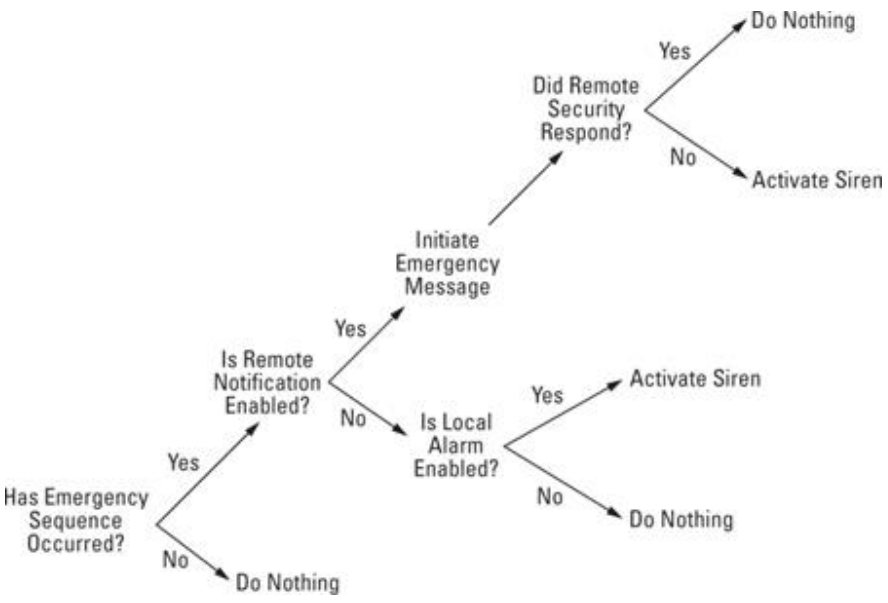
Source: Wikipedia *(en.wikipedia.org/wiki/Decision_table)*

		Rules							
Conditions	Printer does not print.	Y	Y	Y	Y	N	N	N	N
	A red light is flashing.	Y	Y	N	N	Y	Y	N	N
	Printer is unrecognized.	Y	N	Y	N	Y	N	Y	N
Actions	Check the power cable.			X					
	Check the printer-computer cable.	X		X					
	Ensure printer software is installed.	X		X		X		X	
	Check/replace ink.	X	X			X	X		
	Check for paper jam.		X		X				

Alternatively, a decision tree can be drawn to portray decisions in a more graphical form.Figure 18-5 shows a decision tree used to describe a hypothetical emergency sequence.

Figure 18-5. Example of a graphical decision tree

Source: *Managing Software Requirements: A Unified Approach* [Leffingwell and Widrig 2000]


Has Emergency Sequence Occurred?
Yes
No
Do Nothing
Is Remote Notification Enabled?
Yes
No
Initiate Emergency Message
Did Remote Security Respond?
Yes
Do Nothing
No
Activate Siren
Is Local Alarm Enabled?
Yes
Activate Siren
No
Do Nothing

Finite State Machines

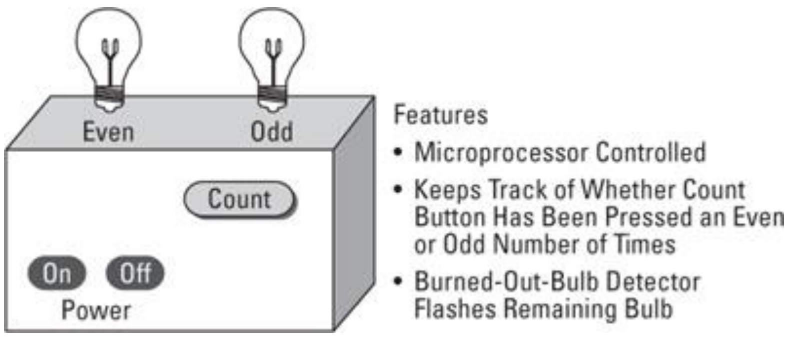
In some cases, it's convenient to model a complex system as a "hypothetical machine that can be in only one of a given number of 'states' at any specific time" [Davis 1993]. In response to an input, such as data entry from the user or an input from an external device, the machine changes its state and then generates an output or carries out an action. Both the output and the next state can be determined solely on the basis of understanding the current state and the event that caused the transition. In that way, a system's behavior can be said to be deterministic; we can mathematically determine every possible state and, therefore, the outputs of the system, based on any set of inputs provided.

Hardware designers have used finite state machines (FSMs) for decades, and a large body of literature describes the creation and analysis of such machines. Indeed, the mathematical nature of the FSM notation lends itself to formal and rigorous analysis so that the problems of consistency, completeness, and ambiguity can be largely mitigated using this technique.

Let's suppose that we have a light box with two lights (Even and Odd) and three buttons, On, Off, and Count, as shown in Figure 18-6.

Figure 18-6. Light box

Source: *Managing Software Requirements: A Unified Approach* [Leffingwell and Widrig 2000]

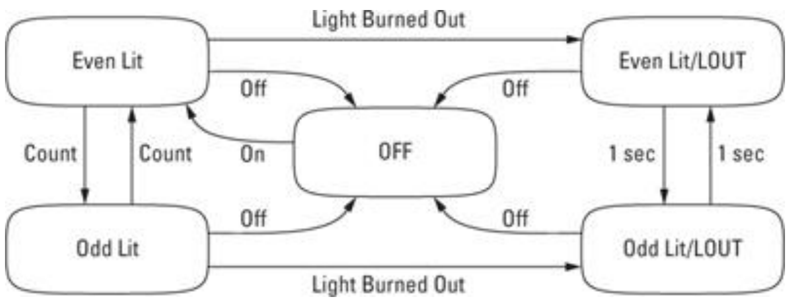

Even
Odd
Count
On
Off
Power
Features
• Microprocessor Controlled
• Keeps Track of Whether Count Button Has Been Pressed an Even or Odd Number of Times
• Burned-Out-Bulb Detector Flashes Remaining Bulb

In natural language, we could express the desired story thusly [Davis 1993, Leffingwell and Widrig 2000].

- After On is pushed but before Off is pushed, system is termed "powered on."
- After Off is pushed but before On is pushed, system is termed "powered off," and no lights shall be lit.
- Since the most recent On press, if Count has been pressed an odd number of times, Odd shall be lit.
- Since the most recent On press, if Count has been pressed an even number of times, Even shall be lit.
- If either light burns out, the other light shall flash every one second.

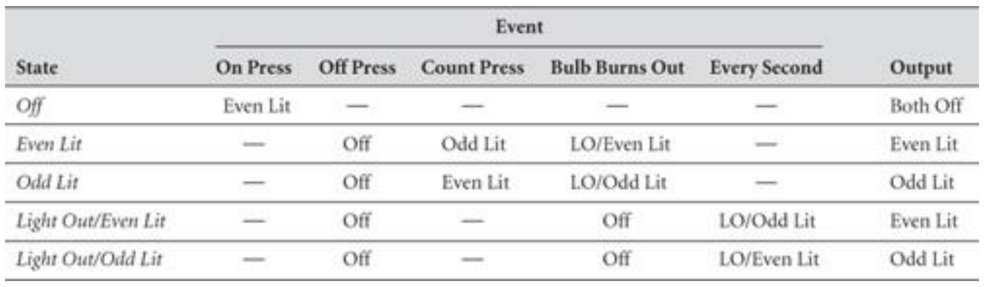
A popular notation for FSMs is the state transition diagram (Figure 18-7). In this notation, the boxes represent the state the device is in, and the arrows represent actions that transition the device to alternative states. We might note that the natural-language expression listed previously of “the other light shall flash every one second” is ambiguous. The state transition diagram in Figure 18-7 is not ambiguous and it illustrates exactly what the product owner desired. If a bulb burns out, the device alternates between attempting to light the Even light and attempting to light the Odd light, each for a period of one second.

Figure 18-7. Example of a state transition diagram



An even more precise form of representing an FSM is the state transition matrix, which is represented as a table that shows every possible state the device can be in, the output of the system for each state, and the effect of every possible stimulus or event on every possible state. This ensures a higher degree of specificity, because every state and the effect of every possible event must be represented in the table. Figure 18-8 defines the behavior of our light box in the form of a state transition matrix.

Figure 18-8. Example of a state transition matrix



	Event					
State	**On Press**	**Off Press**	**Count Press**	**Bulb Burns Out**	**Every Second**	**Output**
Off	Even Lit	—	—	—	—	Both Off
Even Lit	—	Off	Odd Lit	LO/Even Lit	—	Even Lit
Odd Lit	—	Off	Even Lit	LO/Odd Lit	—	Odd Lit
Light Out/Even Lit	—	Off	—	Off	LO/Odd Lit	Even Lit
Light Out/Odd Lit	—	Off	—	Off	LO/Even Lit	Odd Lit

With this technique, we can resolve additional ambiguities that may have been present in our attempt to understand the behavior of the device.

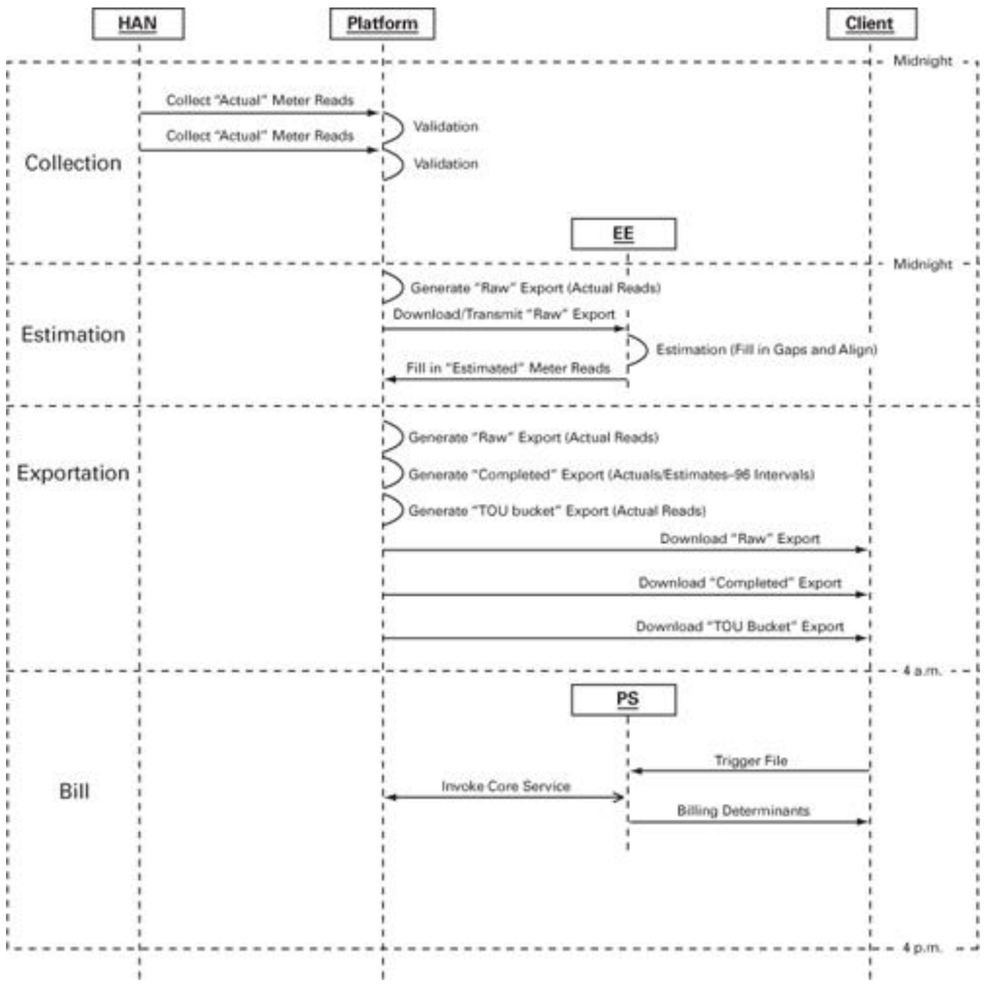
• What happens if the user presses the On switch and the device is already on? Answer: Nothing.
• What happens if both bulbs are burned out? Answer: The device powers itself off.

FSMs are popular for certain categories of systems programming applications, such as message-switching systems, operating systems, and process control systems. FSMs also provide a rigorous way to describe the interaction between an external human user and a system (consider, for example, the interaction between a bank customer and an automated teller machine when the customer wants to withdraw money). However, FSMs can become unwieldy, particularly if we need to represent the system's behavior as a function of several inputs. In such cases, the required system behavior is typically a function of all current conditions and stimuli rather than the current stimulus or a history of stimuli.

Message Sequence Diagrams

A message sequence diagram is a kind of interaction diagram that shows how processes operate with one another and in what order. Message sequence diagrams (MSDs) are a convenient way to express a transactional relationship between two or more parties. Typically, MSDs are used to express relationships such as "A sends this message to B. B responds with this message to A." Figure 18-9 illustrates a typical MSD from the Tendril case study. MSDs identify the interested parties (subsystems in this case) across the top of the diagram, and the interactions proceed down the diagram as time evolves. In this case, the HAN initiates the action, and the messages flow from there.[1]

Figure 18-9. Case study example message sequence diagram


HAN
Platform
Client
Midnight
Collect "Actual" Meter Reads
Validation
Collect "Actual" Meter Reads
Validation
Collection
EE
Midnight
Generate "Raw" Export (Actual Reads)
Download/Transmit "Raw" Export
Estimation
Estimation (Fill in Gaps and Align)
Fill in "Estimated" Meter Reads
Generate "Raw" Export (Actual Reads)
Exportation
Generate "Completed" Export (Actuals/Estimates-96 Intervals)
Generate "TOU bucket" Export (Actual Reads)
Download "Raw" Export
Download "Completed" Export
Download "TOU Bucket" Export
4 a.m.
PS
Trigger File
Bill
Invoke Core Service
Billing Determinants
4 p.m.

Limitations of MSDs[2]

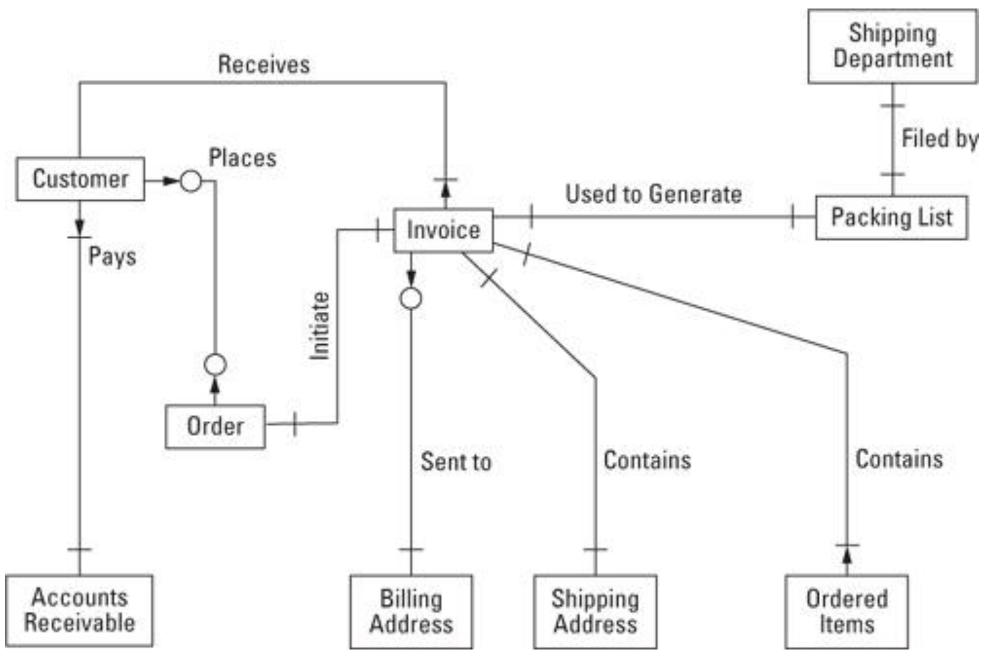
Some systems have simple dynamic behavior that can be expressed in terms of specific sequences of messages between a small, fixed number of objects or processes. In such cases, MSDs can completely specify the system's behavior. Often, behavior is more complex, such as when the set of communicating objects is large or highly variable, when there are many branch points (for example, exceptions), when there are complex iterations, or when there are synchronization issues such as resource contention. In such cases, MSDs cannot easily describe the system's behavior, but they can specify typical use cases for the system, small details in its behavior, and simplified overviews of its behavior.

Entity-Relationship Diagrams

If the stories within a set involve a description of the structure and relationships among data within the system, it's often convenient to represent that information in an entity-relationship diagram (ERD). Figure 18-10 shows a typical ERD.

Figure 18-10. Example of an entity-relationship diagram

Source: *Managing Software Requirements: A Unified Approach* [Leffingwell and Widrig 2000]



Note that the ERD provides a high-level "architectural" view of the data represented by customers, invoices, packing lists, and so on; it would be further augmented with appropriate details about the required information to describe a customer. The ERD does correctly focus on the external behaviors of the system and allows us to define such questions as "Can there be more than one billing address per invoice?" *Answer:* No.

Although an ERD is a capable modeling technique, it has the potential disadvantage of being difficult for a nontechnical reader to understand.

Use-Case Modeling

So far, our agile requirements story hasn't introduced use cases, which are a traditional way to express system behavior in complex systems. Use cases are the primary means to represent requirements with the UML. They are well described there as well as in a variety of texts on the subject. Use cases can be used for both specification and analysis. They are especially useful when the system of interest is in turn composed of other subsystems. But we won't ignore them any longer, because they are the entire subject of the next chapter.

Summary

Agile development avoids big, up-front design (BUFD) and analysis wherever possible. However, your system still has to work and deliver the requisite reliability. When user stories and natural language aren't good enough, your team will need to apply more technical methods to reduce the risk of misunderstanding and to provide additional safety, security, and reliability for your system.

In this chapter, we introduced a number of specification techniques that can reduce ambiguity in specifying system behavior. In general, these technical methods should be used sparingly, and common sense should guide the decision as to which formal technique will be used in a project. If you're building a pacemaker or nuclear reactor control system, perhaps every aspect of the system is critical; in most systems, however, it's unlikely that more than 10 percent of the stories will require this degree of rigor. Choose the method that suits your team best, and apply it only where it is really needed.