Software Product Requirements

Before we can design and build a software product, we must know what it is we are trying to build—or at least, we must know enough about it to begin building it, even though many details may still not be clear. This chapter discusses requirements for software products, which we will often just call *software requirements*, or more often simply *requirements*.

A **requirement** is a property or behavior that something must exhibit—it is something demanded or necessary. A **specification** is a precise description of something. Thus a **requirements specification** is a precise description of properties or behaviors that something must have. For example, a credit card company's web site must show users how much their balance is—the software product *must* display this information, so this is a requirement. A statement to this effect, such as "The web site must display the user's current balance" is a requirements specification. Usually we will not take pains to distinguish requirements from the specifications that express them, and this distinction is often blurred or ignored when requirements are discussed, but you need to understand it.

Required According to Whom?

Requirements are statements about characteristics that a product must have, but who gets to decide this? It seems clear that the people and organizations affected by a product ought to have something to say about it. A **product stakeholder**, or just a **stakeholder** for short, is anyone affected by a product or its development. Most software products have quite a few stakeholders, including the following.

Customers are people or organizations that pay for a product.

Users are people who directly interact with a software product. Often users are not the ones who pay for software products, but they are often the most affected by them.

Clients are people or organizations on whose behalf software products are created. Clients are customers, users, or both; this is a blanket term for customers and users.

Developers of all kinds, including requirements engineers, designers, programmers, and testers, as well as the people who manage and support the developers, are all stakeholders.

Regulators of all kinds responsible for ensuring that software products meet various standards are stakeholders. For example, governments typically have all sorts of regulations to which software products and their development must conform.

Marketers act as stand-ins for clients when mass-market products are created. Marketers also often represent clients when they cannot participate closely in product development.

Even the public can be a stakeholder. For example, software for flying planes or assisting air traffic controllers can affect almost anyone if it results in airplane crashes. Government regulators usually represent the interests of the public.

Stakeholders have a say in the requirements for a product, but they can't have the final word, for several reasons. First, various stakeholders are almost guaranteed to have conflicting desires and ideas about a new product. For example, consider the requirements for an online college course registration system. Students (one kind of user) might want a system that makes it easy to see who has registered for classes so that they can take classes with their friends, classes at the same time as other students who can give them rides to campus, and so forth. The registrar (as a regulator) has to ensure student privacy, which includes class enrollments, so the registrar does *not* want students to see who is enrolled in classes. These sorts of conflicts are frequent.

A second reason that stakeholder desires cannot simply be treated as requirements is that they are invariably incomplete and sometimes even incorrect. Stakeholders typically have vague ideas about what they want a system to do, but very rarely do they have lists of specific properties, characteristics, and features produced with thoroughness and care. Producing such lists is hard work, takes a long time, and employs techniques to help ensure that nothing important is missed. Furthermore, stakeholders often think they understand what they need a software product to do, but it turns out they are wrong. People are often mistaken about their own business processes and what their co-workers actually do. It is not unusual for clients to learn a lot about their own organization and its processes when they help developers produce requirements.

A third reason that stakeholders cannot have the final say about requirements is that their desires and ideas tend to be too abstract to serve as the basis for software design and implementation, and stakeholders typically do not have the necessary knowledge and skills to add the needed details. For example, suppose all stakeholders for a healthy lifestyle web site agree that users must be able to enter information about themselves (like sex, height, weight, blood pressure, exercise habits, etc.) and in response receive recommendations about how to live a healthier lifestyle. But how should the web page be laid out? How should the prompts and responses be worded? What should the steps in the user interaction be? All of this must be determined if code is to be written, so these are vital requirements. Settling these questions calls for user interaction design expertise that few stakeholders possess.

In summary, stakeholders clearly have a say in requirements, but they cannot have the final say. We can clarify the situation by distinguishing between *stakeholder needs* and *requirements*. A **stakeholder need** (or just a **need**) is a feature, function, or property needed or desired by one or more stakeholders. Stakeholder needs reflect what various stakeholders think the product should be and what it should do. As noted above, a collection of stakeholder needs will typically be inconsistent, incomplete, abstract, and sometimes will reflect an incorrect understanding of the situation. Stakeholder needs must be adjusted, augmented, refined, and corrected to produce software product requirements acceptable to stakeholders that can serve as the basis for development.

Like requirements, needs are expressed in detailed descriptions, or **needs specifications**. As noted, often needs are quite vague or abstract, so needs specifications may be descriptions without a lot of detail, but we call them specifications anyway to keep our terminology uniform.

Developers are responsible for transforming needs into requirements and getting all the stakeholders to agree to them. In traditional development processes, certain developers specialize in this job and have job titles such as *requirements analyst*, *requirements specialist*, or *user interaction designer*. In agile methods, teams are cross-functional, so transforming needs to requirements may be done by anyone on the team. Nevertheless, many agile teams have members specializing in this job.

A collection of stakeholder needs poses the problem that requirements analysts must solve. Specifically, the requirements analyst must formulate requirements that resolve the conflicts and correct the errors in stakeholder needs in a way that satisfies stakeholders as much as possible and specifies a good product in sufficient detail to be implemented within the project budget and schedule by the developers available. Thus a collection of needs specifications describes a problem, and a requirements specification describes a solution to this problem.

To recap, stakeholder needs are features or properties that particular stakeholders desire in a product, and requirements are product features or properties agreed by all. Developers work with stakeholders to understand their needs and transform them into requirements that all stakeholders agree to. Who sets product requirements? Stakeholders set the requirements, but only with the help and guidance of product developers.

Functional and Non-Functional Needs and Requirements

Requirements specifications must describe all sorts of things about software products if the right product is to be created by its developers. Commonly, needs and requirements are divided into two broad categories, as follows.

> **Functional needs and requirements** are about how a software product must map inputs to outputs. In other words, functional specifications stipulate the *behavior* of a software product. This category is very broad—it includes such things as the appearance and actions of the user interface; the way data is read, modified, stored, transformed, and output; what a program does in response to the passage of time; and so forth. The bulk of product requirements are functional requirements.
>
> **Non-functional needs and requirements** are about *properties* that a software product must have. Non-functional specifications include stipulations about speed of processing, amount of memory used, communications bandwidth used, frequency of failure, level of security, ease of modification, cost of development, platforms where the product must run, and so on.

Levels of Abstraction in Specifications

Consider the following specifications for a software product.

> The product must provide an online bookstore shopping experience.
>
> The product must provide titles and descriptions of books from which users may choose books to purchase.
>
> The product must allow shoppers to buy books by placing them in a shopping cart.
>
> The product must display a button with each book description that when pressed places books in the shopper's cart.
>
> The product must display a button labeled "Buy" that maintains a constant position (despite scrolling) at the upper right-hand corner of a book-description web page. When pressed, this button must place the book whose description is displayed into the shopper's cart.

Notice that these specifications describe the product at different levels of abstraction, each with more detail than the one before. Note too the huge range of abstraction comprehended by product specifications, from those like the first that encompass entire systems, to those like the last that describe the smallest implementation detail. When working with needs and requirements, developers typically begin with quite abstract specifications for whole products and gradually refine them by adding more and more detail until they are specific enough to implement in programs.

Software engineers have terms for specifications at various levels of abstraction. **Business requirement specifications** are statements of client or development organization objectives that a product must meet. For example, the first specification above states a business requirement. Other examples include the following.

> The initial version of the product must be delivered by May first of this year.
>
> The product must generate sales of one million dollars during its first year of operation.
>
> The product must reduce the cost of order fulfillment by half within a year of delivery.

Business requirements tend to be very abstract, particularly about what a product actually does, but more specific about what benefits the product is expected to provide for the organization or the client. Business requirements usually are based on needs from marketing or management stakeholders. They also tend to be non-functional requirements. In some sense business requirements are often hardly software requirements at all because they specify characteristics not definitive of particular products, such as sales, profit, or market penetration goals. Nevertheless, they provide context for the software product that may have consequences for more specific requirements.

One rung down on the ladder of abstraction from business requirements are **user-level needs and requirements**. These are about how a product must support user needs (hence the name), though in fact needs of other stakeholders besides users may appear at this level of abstraction as well. Typically user-level needs and requirements indicate that a product must enable users to achieve certain goals or complete certain tasks. Some examples are the following.

> The product must produce earning reports suitable for financial auditing.
>
> The product must allow personnel clerks to set employee salaries and wages.
>
> The product must allow employees and their managers to view customer evaluations of their performance.
>
> The product must authenticate all users and assign them roles governing access to information.

Notice that user-level specifications may still be quite abstract, but they do indicate particular product properties or behaviors. Notice too that they may be functional or non-functional specifications.

Operational-level needs and requirements are about individual inputs, outputs, computations, operations, characteristics, or properties of a product. These can be quite specific. The following are examples of operational-level specifications.

> The product must allow users to enter polynomial equations, trigonometric equations, logarithmic equations, and exponential equations, all of one variable.
>
> The product must support the six standard trigonometric function, logarithmic functions with arbitrary bases, and exponential functions with arbitrary bases.
>
> The product must display two-dimensional graphs of entered equations.
>
> The product must display a graph of any well-formed equation in no more than a fifth of a second.

Finally, we can distinguish **physical-level needs and requirements**. These are about details of the physical layout and behavior of the product's user interface, the format of input and output data, and other low-level details. The following are examples of physical-level specifications.

> The product must provide a text-box for equation input. The text box must display 50 characters and scroll horizontally up to 800 lines.
>
> The product must support cut-and-paste of text into all text boxes.
>
> The product must display numbers rounded to three decimal places with terminal zeros truncated. If a number has no decimal digits then the decimal point must not be displayed.

There are no hard and fast distinctions between levels of abstraction in specifications. There is nothing to be gained by trying to classify particular examples of requirements or needs statements as being user-level, operational-level, or physical-level specifications. We distinguish levels of abstraction in specifications to illustrate and emphasize the point that specifications are stated across a broad range of levels of abstraction. Furthermore, needs and requirements are usually conceived and stated at high levels of abstraction and refined over time to lower levels of abstraction. This is an essential, important, and expensive task in software development requiring considerable skill and quite a lot of time. We discuss refinement further below.

It is also important to appreciate that requirements are realized at the physical-level even if they are never explicitly refined to that level. It may be the case, for example, that no stakeholders particularly care exactly how many characters are displayed in a text-box, or what font is used in a button label, and so no one bothers to include such things in the requirements for the product. But all these details must be decided when software is actually written, so someone (perhaps the programmer when she is actually writing the code) *will* decide on all requirements at the lowest level of abstraction.

Requirements in Traditional Processes

The framework and terminology we have been discussing so far reflect the traditional approach to software requirements that has been used in software engineering for decades. The context for this approach is some form of the waterfall life-cycle process in which the work for a particular phase of development is mostly done at one time. The first thing that one needs to do is understand what to develop, so stakeholder needs and product requirements are determined first. In the waterfall model, *all* the requirements for the product (or as near as possible) are determined before almost anything else is done. Once set, (or **frozen**, as is typically said) requirements are difficult and expensive to change, so requirements change is resisted.

Unfortunately, this has often not worked well, for the following reasons.

- It is very difficult (perhaps impossible) to determine all requirements for non-trivial products before any implementation is done. Stakeholders have a hard time even figuring out their needs without having some idea of what the product will do and without seeing early versions of it. Analysts likewise find it difficult to choose requirements without exploring alternatives as they are realized in the product.
- The world changes rapidly, and so do stakeholder needs and desires. Making requirements impossible or hard to change leads to products stakeholders don't want when they are delivered.
- Writing all the requirements for a significant product is an enormous and difficult job, but it must be done if developers won't be implementing the product until well after requirements are formulated—there is no way that anyone could remember so many details. Hence waterfall approaches necessitate large, unwieldy requirements documents that cost a lot of time and money to produce.
- Following a waterfall process usually means that a product is not delivered for months or years after its development project is started. Many projects are cancelled or never even get started because of this huge delay. Developers also find this delay demoralizing, which damages their productivity.

These problems have posed difficulties for software engineering since its beginning. Many ways of addressing these problems have been tried, such as making heavy use of product prototypes so that stakeholders and analysts can get a better idea of what a product might do and how it might work as they formulate requirements. Another approach has been to develop products in smaller increments so that the time between project start and product delivery is reduced; a product with minimal function is delivered early and then evolved into a full-featured product over time with several releases.

Requirements in Agile Processes

Agilists try to avoid writing traditional software requirements specifications altogether as a way of solving the problems with traditional requirements specifications discussed above. However, as noted, requirements *must* be present, even if only implicitly, before code can be written. Scrum uses several approaches to make requirements specification and change as easy and cheap as possible, and we will discuss them now.

Scrum projects have stakeholders with needs and desires just like all other projects, and they begin with business requirements like other projects. But rather than proceeding from these to user-level requirements that are then elaborated into operational-level and then physical-level requirements recorded in a large requirements specification document, Scrum projects have lists, called *product backlogs*, that guide development. Product backlog items are added based on stakeholder needs and desires. A person called the **product owner** acts as a stand-in for the product's stakeholders and adds backlog items and sets their priorities. The highest priority items are selected for implementation in each development iteration, or **sprint**. Product backlog items are added, removed, changed, and reprioritized by the product owner continuously, thus altering the direction of development. The fluidity and ease of change of the product backlog is a large part of the justification for calling Scrum an "agile" method.

Items added to the product backlog capture needs and desires of various stakeholders, so they express stakeholder needs. Product backlog items are prioritized in terms of the value they add to the product. Thus a **product backlog item** (**PBI**) is a statement of a stakeholder need, and the **product backlog** is a prioritized list of unimplemented stakeholder need specifications. When a PBI is selected for implementation in a sprint, it is because the item has high priority and the decision has been made that it is time to incorporate it into the product. This decision is tantamount to the determination that a stakeholder need is of sufficient importance, and that its inclusion in the product is sufficiently valuable, that the need should become a requirement. PBIs selected for implementation go into a **sprint backlog**. Thus a **sprint backlog item** is a requirements specification.

With this brief introduction to the Scrum product backlog, we can catalog ways that Scrum makes requirements specification and change as easy and cheap as possible.

Delay choosing requirements as long as possible—Stakeholder needs can be added to or removed from the product backlog, or changed, on a whim. Product requirements, however, are only set once PBIs are selected for implementation. It is easy and cheap to manipulate needs statements, but harder to change requirements because they represent important product design decisions. In Scrum requirements are chosen just before they are implemented, and they cannot be changed until a sprint is completed. Delaying choosing requirements avoids the cost of changing them.

Delay refinement as long as possible—PBIs are initially stated at a high level of abstraction, at about the user-level. Such high-level items typically describe product features that cannot be implemented in a single sprint. These large PBIs are refined into smaller and more detailed PBIs until they are small enough to be implemented in a sprint, at a level of abstraction that roughly correspond to operational-level needs. User-level PBIs are not refined until just before they are expected to be implemented in a sprint, so needs refinement at this level of abstraction is delayed as much as possible. Finally, some operational-level needs are selected for implementation during a sprint, thus becoming product requirements. At this point, developers interact with stakeholders to refine these operational-level requirements to the lowest-level of abstraction, physical level. Again, refinement is delayed as much as possible.

Delaying specification refinement avoids much rework and speeds product delivery. Refinement occurs as late as possible, so product delivery is not delayed by (perhaps unnecessary) requirements analysis work. Needs in the product backlog can change at any time—if a PBI is removed or altered, the work done to refine it need not be redone because it was (usually) never done in the first place—user-level needs are refined only just before they are implemented, and operational-level requirements are refined only when they are implemented. Of course, an implemented requirement may ultimately be rejected and all the work done refining and building it wasted, but this is (ideally) a relatively rare occurrence.

Avoid writing requirements altogether—In Scrum and most other agile methods, operational-level PBIs are written down, but physical-level requirements are usually not. When a PBI is chosen for implementation in a sprint, developers converse with stakeholders and the product owner to understand their needs and then make decisions about detailed requirements. Discussions with stakeholders and the product owner about detailed requirements may be extensive, and sometimes result in written physical-level requirements, but usually developers just go ahead and implement the results of their discussions immediately in the sprint. The bulk of traditional requirements documents are detailed (often physical-level) requirements specifications. By not writing such requirements, agile developers avoid a great deal of requirements documentation work.

Determine requirements in light of current product features—As noted, stakeholders have trouble identifying their needs and desires and developers have trouble choosing requirements without seeing how they might fit into the product. In Scrum, stakeholders see the latest product increment at the end of every sprint in a sprint review. Stakeholders and developers can then consider needs and requirements in light of the latest version of the product. Feedback from stakeholders is used by the product owner to alter the product backlog and prioritize PBIs, and also provides input to decision about what PBIs to implement in the next sprint. Thus the Scrum process provides stakeholders and developers with product information to guide them in their choices of needs and requirements.

In summary, Scrum tackles the problems of traditional waterfall approaches to requirements specification by (a) delaying choosing requirements as long as possible, thus avoiding requirements change rework; (b) delaying refinement as much as possible, thus speeding product delivery and facilitating product change by avoiding rework, (c) not writing detailed requirements specifications, thus speeding delivery and avoiding work, and (d) providing product increments that help stakeholders identify needs and developers analyze requirements. Studies have shown that these practices are effective in solving the problems resulting from writing large requirements specifications early on.

Stating Specifications in Traditional Processes

Traditional waterfall-style process needs and requirements specifications are stated in declarative English sentences. Despite the availability of several mathematical specification languages (such as Z [1], for example), most specifications are stated in English (or other natural languages) so that they are easy for everyone to read. Unfortunately, English specifications may be vague or ambiguous, and still hard to understand. For example, consider the following specification.

The product must draw parts with and without notations showing sub-assemblies.

Does this mean that part drawings can include notations, and the notations show sub-assemblies, or that the part drawings show sub-assemblies and in addition may have notations? Dividing it into several simpler specifications, rewording it, and defining its terms can disambiguate this specification.

We try to make requirements specifications clear and unambiguous by following rules of good technical writing, such as the following.

- Write complete, simple sentences in the active voice.
- Define terms clearly and use them consistently.
- Use the same words for the same thing—avoid synonyms.
- Group related material into sections.
- Use tables, lists, indentation, white space, and other formatting aids to present and organize material clearly and concisely.

Following these rules makes specifications easier to find, understand, and change.

In addition, we follow some rules particular to specification writing that help make them easier to understand, refer to, verify, and validate. The first of these rules is to express all specifications using the words "must" or "shall." This reflects the fact that needs and requirements are statements about features or characteristics that a product must have. It also helps distinguish specifications from other statements that provide additional information or state features or characteristics that a product *may* but need not have. If you look back over the examples presented so far, as well as those below, you will see that all of them use "must," so they all serve as examples of this convention.

Another rule about writing specifications is that they should be **testable** or **verifiable**, which means that each specification should be written in such a way that there is a definitive procedure for determining whether it has been satisfied. To illustrate, the following requirement specifications are not testable.

> The product must display query results quickly.
>
> The product must produce error messages that are easy to understand.
>
> The product must be able to handle several users at once.

How would a tester determine whether results are displayed quickly enough, or that a product can handle enough users at once? Testable requirements include clear and measurable properties that can be used to determine whether the requirement is satisfied. The requirements above could be restated as follows.

> The product must display query results in less than one second.
>
> The product must produce error messages that (a) state the problem without referring to implementation details known only to developers, (b) state whether the program can continue and, if it can continue, the consequences of continuing, and (c) explain how to get help responding to the error.
>
> The product must be able to handle up to 18 users at once.

Specifications should be easy to understand and to change. It should also be easy to associate individual requirements with parts of design that realize them, code that implements them, tests that verify them, and so forth. This latter activity is called **requirements traceability**. Specifications that state several needs or requirements are harder to understand, harder to change, and difficult to associate with other artifacts for traceability. Thus another rule for writing good specifications is that each specification should state only a single need or requirement. Such specifications are called **atomic**. For example, consider the following specifications.

> The product must display a list of previous commands and the results of commands, each in its own window.
>
> If the user presses an up or down arrow key or selects a previous command with a pointer device, then previous commands must be scrolled through or the selected command displayed.

The first specification clearly states two needs or requirements. The second also states two needs or requirements, and conflates them so that neither one is clear. The following atomic versions of these specifications are better.

> The product must display a list of entered commands in an entered commands window.
>
> The product must display the result of all commands in a results window.
>
> The user must be able to scroll through previous commands in the command textbox using the up and down arrow keys.
>
> The user must be able to insert a previous command into the command textbox by selecting it in the previous commands window using a pointer device.

A final rule about specifications, and especially requirements, is that each one should have a unique identifier. Needs and requirements identifiers make it easier to refer to them in discussions and when tracing requirements. A typical form for identifiers is a dotted-number scheme that uses requirements document sections. For example, 4.2.8.3 might be the identifier for the third requirement in chapter 4, section 2, subsection 8 of the requirements document. Assigning every requirement its own identifier makes it easier to refer to requirements in other parts of the requirements specification document, or in other documents.

Despite all these rules and recommendation about writing requirements, it is simply the case that writing unambiguous, detailed, concise, and complete requirements is very difficult, and furthermore that even the best-written requirements are easy to misunderstand. Both requirements writers and readers must work hard to express and understand requirements specifications, being willing to rewrite and reread them over and over again.

Stating Specifications in Agile Processes

As noted before, agile developers try to avoid writing requirements specifications. But stakeholder needs and requirements specifications are still written down in various documents, like product vision statement, and certainly as product and sprint backlog items. These specifications may be written in the traditional manner as discussed above. However, agilists have almost universally adopted alternative means of stating specifications, which we consider now.

Agile methods usually express functional needs and requirements in user stories. A **user story** is a description of a function a product must provide for a user. User stories can be expressed at many levels of abstraction. At the highest level of abstraction, user stories describe functions that cover a large fraction of an entire application, or perhaps even an entire application, and would take many months or years to implement. Such stories are called **epics**. User stories that would take several sprints to implement (at about the user-level of abstraction) are called **features**. User stories small enough to be implemented in a sprint (at the operational-level of abstraction) are **sprintable stories** or **implementable stories**. Some Scrum experts prefer to restrict the term **user story** to just implementable stories, but we will use the broader meaning of this term here.

User stories are sometimes expressed simply as lists of functions that a product might provide. For example, a product vision statement might include the following in its product description.

> Display drawings of individual parts.

Display drawings of assemblies with exploded views.

Rotate drawings in any direction.

Zoom in and out of drawings.

Such lists can be interpreted as specifications by understanding them as lists of functions that a product must provide.

There are two shortcomings of simply listing functions. The first is that when they express stakeholder needs, they do not identify the stakeholders. The second problem is that they are typically not testable. This is not a problem for abstract needs statements, but it becomes more of a problem as specifications become more detailed.

The most widely used way of expressing user stories is called the **user voice form.** It has the following format.

As a <*role*>, I want to <*activity*> so that <*benefit*>.

The <*role*> is filled in by a **user role**, which is a name for a class of product users. The <*activity*> is filled in by a function performed by a system. The <*benefit*>, which is optional, specifies the value of the activity for that class of users. The following examples illustrate this form of user story.

As a payroll clerk, I want to enter salary data so that payrolls will use adjusted salaries.

As a design engineer, I want to display an expanded parts drawing so that I can understand how parts fit together.

As a bank teller, I want to see account balances so that I can answer customer inquiries.

As a department head, I want to edit project portfolios.

If all product backlog items are user stories, how does a stakeholder who it not a user put a need into the product backlog? One solution is to broaden the <*role*> entry in user stories to include all stakeholders rather than just user roles. Another is to abandon the user voice form. Another is to allow product backlog items that are not user stories. Any of these alternatives may be used, but here we elect the first and so characterize the user voice form as naming a stakeholder, the product function that stakeholder desires, and why the stakeholder desires it.

The user voice form provides information about stakeholders, but these statements may still not be testable. Scrum experts recommend that user stories, in particular sprintable user stories, also have **acceptance criteria** or **conditions of satisfaction** stating the product behavior regarded as fulfilling the user story. Acceptance criteria should be specific enough that it will be clear whether the product fulfills the user story. Acceptance criteria are a vehicle for adding details to somewhat abstract user story specifications.

Agile methods often employ the "three Cs of user stories:" card, conversation, and confirmation.

Card—A card is an index card or post-it note with a user story title and a statement of the user story on one side. Typically user stories employ the user voice form, but additional information may be included as well, though part of the point of using a card is to keep the specification short.

Conversation—The card is regarded as a placeholder for various conversations amongst developers and stakeholders to discuss the need when it is modified, refined, estimated, prioritized, and (perhaps) implemented. These conversations may alter the need or fill in details as necessary during development. Many of these details will not be written down because the emphasis in agile methods is on face-to-face discussion rather than documentation.

Confirmation—The back of the card contains the acceptance criteria for the user story. Because it is small, many details may be left out, which again can be derived during conversations and captured in tests.

Actual physical cards can be used and they can constitute the product backlog. Alternatively, the product backlog can be kept in a spreadsheet with "cards" being rows with cells for user stories and acceptance criteria. There are also many agile tools that automate product backlogs. The format of PBIs and the product backlog is not important as long as documentation overhead is minimal and it is easy to add, remove, change, and shuffle PBIs in the product backlog.

There is nothing wrong with supplementing PBIs with additional information either. For example, there may be detailed interface specifications relevant to PBIs about interfaces, or papers about algorithms or implementation strategies connected with certain PBIs. Such information (or references to it), can supplement PBIs as needed.

Eliciting Stakeholder Needs in Traditional Processes

In a traditional development process in which most requirements are determined before further development work is done, eliciting all (or most) stakeholder needs must be done early and thoroughly. The size and importance of this effort have given rise to many techniques that we catalog in this section.

Eliciting stakeholder needs for software products is similar to extracting needs and preferences in many other disciplines. For example, pollsters elicit voter preferences, market researchers try to determine what consumers are willing to buy, and TV ratings researchers try to find out what people like to watch on television. Techniques used in these disciplines are sophisticated and highly refined; we can only survey them here. These techniques have been used widely in traditional development; agile developers do things somewhat differently, as we discuss below.

The following are the most frequently used techniques used to elicit stakeholder needs in traditional processes.

Interviews—An **interview** is a question and answer session. Developers ask stakeholders what functions and characteristics they need, and record the answers. The developers are trying to understand the stakeholders' perspectives, so usually it helps to also ask stakeholders to explain the domain of application, their vision for the product, how they think it could be valuable, and so forth. Some questions should be prepared before the interview, but it is valuable to allow interviewers to formulate questions during the session as new ideas and issues arise. Answers should be recorded. It is much easier to record the session on audio or video recording equipment than to try to take notes.

Observation—If a product supports users in doing particular tasks, or is intended to replace a system currently in place, then it is often useful for developers to watch users doing the tasks or using the system to be replaced. This is called **observation**. It allows developers to understand the context and goals of the product, and it provides a wealth of information leading directly to detailed requirements. Developers should watch people do the same task several times, and watch several people do the same task so they understand how its performance can vary. It is easier to record subjects with videotape than to take notes. Subjects can be asked to explain what they are doing as they are doing it. This technique, called the *think-aloud protocol*, enables observers to understand better what subjects are doing, though it may also distort what the subject actually does.

Focus groups—A **focus group** is an informal discussion among six to nine people led by a facilitator who keeps the discussion on topic. Stakeholders can meet to discuss the product and its features and characteristics led by a developer or a trained facilitator. Other developers can observe the sessions or learn about them second hand. Focus groups are a good way to collect a broad spectrum of ideas and perspectives, with feedback from other stakeholders, in a short period of time. They are especially useful early in needs elicitation.

Workshops—A **workshop** is a meeting whose participants focus intently on a particular topic or goal. Elicitation workshops are meetings with stakeholders aimed at discovering and documenting stakeholder needs and desires. Workshops require effort from their participants, but they can be very productive.

Prototypes—A **prototype** is a working model of part or all of a final product. We have noted how difficult it is for stakeholders to decide what products should do and how they should work. Prototypes give stakeholders a chance to try out different versions of a product. This is usually very helpful in nailing down good requirements.

Document studies—Developers can learn about a product problem domain, the needs of stakeholders, product constraints, and so on, by studying various documents describing the business, its policies and procedures, quality improvement studies, bug reports, suggestions for improvement, and so forth. Studying documents cannot by itself provide everything developers need to know, but it can provide a lot of the context necessary to really understand stakeholders.

Competitive product studies—If a product has competitors already on the market, analyzing them is a prerequisite for generating requirements for a new product. Obviously, new products must avoid the weaknesses and match the strengths of existing products or they won't be successful.

Requirements specialists usually use several of these techniques over the course of the requirements analysis phase of the traditional life cycle, often returning to stakeholders for follow-up interviews or observations, demonstrations of new prototypes, and so forth, as requirements are developed over time.

Eliciting Stakeholder Needs in Agile Processes

Agile methods do not attempt to determine most requirements early, so the need to elicit all stakeholder needs early goes away. Instead, needs are elicited continuously during development. The main advantage of this approach is that stakeholders can express needs that have only just arisen in response to the current situation. Also, stakeholders can see the current state of the product and react to it in formulating needs. In effect, the product itself becomes a continuously evolving prototype. This usually makes it easier for stakeholders to understand the product, and helps ensure that needs to not become obsolete. But there are drawbacks too, particularly the following.

- Agile methods do not advocate systematic means for eliciting needs (such as those listed above). Stakeholders may over-react to recent events in stating and prioritizing needs, possibly losing track of the big picture and making poor decisions about what to include in the product.
- Agile methods usually place great trust in a few stakeholders or stakeholder representatives. In Scrum, for example, the product owner has primary responsibility for the product backlog, with input from whichever stakeholders make time to attend sprint reviews or make their opinions known in other ways. There is considerable risk that the product will not reflect the needs of all stakeholders, or that important stakeholders' opinions will not be properly taken into account.

There is no reason that agile methods cannot use the several systematic techniques for collecting stakeholder needs used in traditional processes, but these techniques are not part of agile culture.

Verifying and Validating Requirements in Traditional Processes

Requirements specifications must be nearly complete before moving on to implementation in traditional processes, so it is important to ensure that they are right. There are five important characteristics to be checked.

Clarity—If requirements are vague or ambiguous, then questions will arise during development, or perhaps the product will not embody the intentions of stakeholders. So checking for clarity is important. A good way to check clarity is to ensure that requirements are testable.

Consistency—Requirements that contradict one another can obviously not be implemented.

Completeness—Requirements specifications that fail to fully characterize the product will result in a product lacking features or properties that may be important to stakeholders. Requirements also need to be sufficiently detailed.

Correctness—Requirements specifications that do not accurately reflect stakeholder needs will result in a poor product.

Well-formedness—As noted, requirements specifications should be stated using "must" or "shall," should be atomic, and should have unique identifiers.

The main technique for checking requirements specification is **reviews**, which are examinations and evaluations of work product by qualified individuals. There are various kinds of reviews, which we discuss elsewhere, and any of them can be used for requirements. All reviews are expensive because they require several people to perform a painstaking task that usually requires a lot of time. Nevertheless, studies have shown it is less expensive to do reviews and detect and correct errors during the requirements process than to not do them and detect and correct the errors much later in the development process.

Of course the real test of requirements specifications occurs when the product is delivered to its customers. All too often, it becomes clear that important requirements have been missed or misunderstood, but at that point, a great deal of money and time have been spent, and even more would have to be spent to fix the problems. This is the Achilles heal of traditional processes.

Verifying and Validating Requirements in Agile Processes

Requirements are under continuous scrutiny in agile processes. In Scrum, for example, the product backlog is always being groomed, so requirements are always being reconsidered. The most careful check of requirements, however, is during the reviews at the conclusion of each sprint. The team presents the latest version of the product to the product owner and any other stakeholders who care to attend the review. Requirements problems should manifest themselves in the form of wrong, missing, or extraneous product behaviors or properties. These problems can then be traced to erroneous requirements that can be fixed immediately. Stories to alter the product and correct problems can be added to the product backlog.

The greatest weakness of this approach is that requirements errors will not be detected because a stakeholder who could recognize errors is not present at sprint review. Not every stakeholder can attend every sprint review. The product owner stands-in for missing stakeholders, but if the product owner's understanding of stakeholder needs is incorrect or incomplete, then errors may go undetected.

Requirements Management in Traditional Processes

In traditional processes, a project is launched with a **product mission statement** or some other document that provides high-level (business) requirements for a product. One or more requirements specialists then conduct an extensive requirements analysis. **Requirements analysis** is the activity of eliciting, examining, and understanding stakeholder needs, developing requirements specifications, and evaluating them to ensure that they are clear, complete, consistent, correct, and well formed. The output of this activity is a **software requirements specification (SRS)**, a document containing all the requirements for a product. The SRS is usually divided into sections consisting of atomized requirements statements. The figure below shows a template for an SRS.

1. Introduction
    - 1.1 Product Vision
    - 1.2 Project Scope
    - 1.3 Stakeholders
    - 1.4 Design and Implementation Constraints
2. Functional Requirements
    - 2.1 Product Behavior
    - 2.2 User Interfaces
    - 2.3 System Interfaces
    - 2.4 Data Requirements
3. Non-Functional Requirements
4. Other Requirements
5. Glossary

Figure 1: An SRS Template

Sometimes an SRS will also contain tables, figures, and various sorts of models. During its writing the specifications in the SRS will be reviewed and corrected; often there will be a final review at its completion as well. Once it is finished, the SRS is the basis for design, implementation, and testing.

Although great effort is expended to ensure that the SRS is clear, complete, consistent, correct, and well formed, shortcomings in all these areas will inevitably arise during the rest of the development process. Furthermore, stakeholders' needs and desires may change. Hence, the SRS will need to be changed. Usually a special process for requirements modification is instituted to ensure that requests for change are logged, evaluated, and decided upon with due consideration of their importance and costs. Then the changes are made and traced through designs, code, and tests to make sure that the changes are realized and checked. Clearly, this is an expensive endeavor.

The entire activity of developing requirements and then overseeing them throughout the software development process is called **requirements management**.

Requirements Management in Agile Processes

Agile methods do not contain an explicit requirements management process, but we can sketch the role of requirements in agile processes, and in particular in Scrum. Agile projects begin with some sort of project mission statement stating high-level needs in the form of epics or features that become the start of the product backlog. The product owner, usually with the help of the team and other stakeholders, refines these high-level features into user stories and prioritizes them to the point that some of the stories are sprintable. Then the team begins sprinting.

During each sprint, operational-level requirements in the form of user stories are refined into physical-level requirements in conversations with stakeholders and the product owner, and then implemented. At the conclusion of each sprint, stakeholders and the product owner validate requirements by participating in product demonstrations in the sprint review.

As the team sprints, the product owner continues to groom the product backlog, refining and reconsidering needs. Needs will typically be added, deleted, and modified as the product backlog is changed.

As this summary makes clear, requirements management is a core activity of agile methods, but it is incorporated into the broader processes of backlog grooming and sprinting, and not distinguished as a separate process.

Requirements and Product Design

Suppose you work for an automobile company as an automotive designer. Your job is to come up with ideas for appealing new cars with state-of-the-art features and then specify them in every detail so that they can be manufactured. In the automobile industry (and most similar industries) your job is known as *product design*. If you think about it, you will realize that your automotive design job is very much like the software requirements analyst's. In fact, requirements analysis *is* product design for software. It is not clear why the software industry has adopted terminology so different from other industries, but it has. This is unfortunate for two reasons:

- It hides the fact that software product development is very similar to other kinds of product development. Many product design practices, tools, and techniques in other engineering disciplines are perfectly applicable in software, but this is not obvious because of the difference in terminology. Therefore, software developers do not use many of these practices, tools, and techniques.
- Perhaps because it is called *requirements analysis* rather than *product design*, specifying the requirements of a software product has not been considered a design activity. Being sloppy about distinguishing needs and requirements also encourages people to think that requirements analysis is mostly about gathering stakeholder needs, writing them down, and calling the result a specification. This tendency, which originated in traditional processes, has carried over into agile processes as well: the product owner is really the chief product designer, and team members do product design when working with stakeholders to formulate detailed specifications during sprints. As discussed, there is a lot more to requirements analysis, backlog grooming, and requirements refinement than just writing down stakeholder needs. In particular, good developers must generate alternative designs, consider their relative merits in light of stakeholder needs, improve them if possible, and ultimately settle on the best ones as the requirements. This is a design process. Calling what developers do when pinning down requirements *product design* would not only be more accurate, but would encourage these designers to *do* design, not just gather and document needs.

As an illustration of why recognizing requirements analysis as a design activity would be beneficial, lets consider some of the characteristics of design. A fruitful way of thinking about design is as problem solving: stakeholder needs and project constraints present a design problem, and a product specification is a solution to the problem. Problem solving has several salient characteristics.

Trial and error—Problem solving usually involves a certain amount of trial and error. Rarely are acceptable product specifications created on the first try. Typically several attempts are needed before a good solution emerges.

Iterative—Requirements analysis is an iterative activity. This is another way of saying that it involves trial and error, except now we are focusing on the process. First, one or more tentative solutions are generated, then they are evaluated to determine the advantages and disadvantages of each, and finally a decision is made whether a solution is acceptable. If not, then we return to step one, using what we have learned from previous attempts to generate better solutions.

No unique solution—There is often more than one way to solve a problem. Different analysts may come up with equally good ideas. If it is not apparent which of several specifications is best, then pick one of the good ones and move on (perhaps noting the alternatives in case problems arise with the chosen solution later on).

If requirements analysts, product owners, and agile team members think of themselves (at least in part) as product designers, then they would know to expect to do a lot of iteration as they explore design alternatives, that a lot of their product designs will have to be thrown away, or at least greatly changed, and that there may be several good ways to solve their problems (as well as a lot of bad ways).

References

1. J. M. Spivey, *The Z Notation: A Reference Manual.* Prentice Hall, 1989.