Chapter 3

Agile Requirements for the Team

When you're part of a team, you stand up for your teammates. Your loyalty is to them. You protect them through good and bad, because they'd do the same for you.

—Yogi Berra

Introduction to the Team Level

In the previous chapter, we provided an overview of the basic organization, process, and requirements artifacts model we'll use to implement software agility. Of course, we won't get very far without first understanding the basic nature of the agile team itself and how it organizes its work to deliver the value stream to its customer. In this chapter, we'll elaborate on the Team level of the Big Picture, as illustrated in Figure 3–1.

Why the Discussion on Teams?

One might wonder why a book on software requirements leads with a discussion of the organization, roles and responsibilities, and activities of the agile project *team*. As we described in the previous chapters, the nature of agile development is so fundamentally different from that of traditional models that, by necessity, we must rethink many of the basic practices of software development. For many, adopting

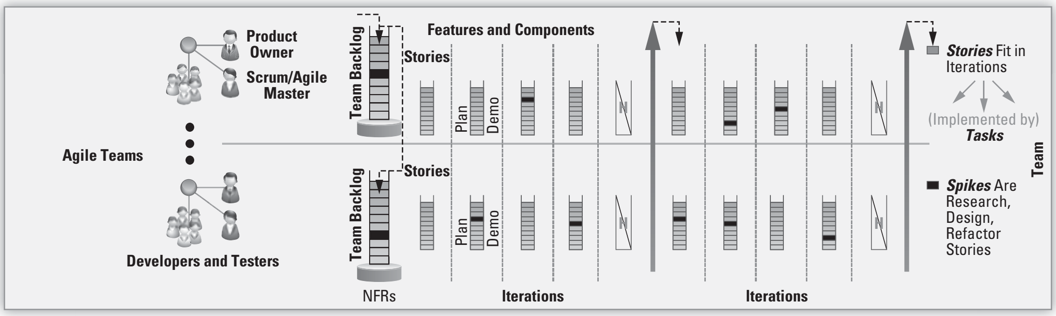

Figure 3–1 The Team level of the Big Picture

agile challenges the existing organizational structure, the assumptions about the relationships among team members, and even personnel reporting structures.

More importantly, in agile, the organization of the requirements and the organization of the team itself are not independent things. No longer do large batches of predetermined requirements defined by others get thrown "over the transom" to a set of developers for implementation—individuals that are organized, via matrix or other, for whatever purposes.

> Rather, the teams organize around the requirements so as to optimize the efficiency of defining, building, and testing code that delivers value to the end users.

The entire team is integrally involved in defining requirements, optimizing requirements and design trade-offs, implementing them, testing them, integrating them into a new baseline, and then seeing to it that they get delivered to the customers. That is the sole purpose of the team.

To understand the team organization challenge better, let's consider the challenge of "producing working code in a timebox" and see what type of organization might best accomplish this.

The basic unit of work for the team is the user story (the topic of Chapter 6). The team's objective is to define, build, and test some number of user stories in the scope of an iteration and thereby achieve some even larger value pile in the course of a release. Each story has a short, incredibly intense development life cycle, ideally followed by long-term residence in a software baseline that delivers user value for years to come. Pictorially, Figure 3–2 shows the life cycle of a story.

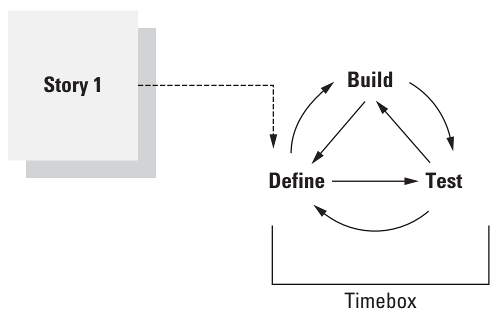

Figure 3–2 Defining/building/testing a user story

At the time of arrival, the story will likely have some amount of elaboration done during one or more prior iterations, or it may simply be a labeled placeholder for a "thing to do that we'll figure out later." At the iteration boundary, however, later is now. In essence, each story operates in the same pattern: Define the story, write the code and the test, and run the test against the code. These are all done in parallel so we call that sequence *define/build/test*. In so doing, we use the sequence as a verb to illustrate the fact that the process is concurrent, collaborative, and atomic. It is done completely, or it isn't *done*. However, even an atom has its constituent parts, and ours are as follows.

- *Define:* Even if the story is well-elaborated, the developer will likely still interact with the product owner to understand what is meant by the story. Also, some design will likely be present in the developer's mind, and if not, one will quickly be created and communicated to the product owner, peer developers, and testers. We use the word *define* to communicate that this function is a combination of both *requirements and design*. They are *inseparable*; neither has any meaning without the other. (If you don't know *how* to do IT, then you don't really know what IT is!)
- *Build:* The actual coding of the story provides an opportunity for new discovery as well. Conversations will again ensue between developer/product owner, developer/other developers, and developer/tester. Story understanding evolves during the coding process.
- *Test:* A "story" is not considered complete until it has passed an acceptance test (the topic of Chapter 10), which assures that the code meets the intent of the story. Building functional acceptance tests (plus unit tests) before, or in parallel with the code, again tests the team's understanding of the subject story.

Of course, this process happens every day; it happens in real time, and it happens multiple times a day for each story in the iteration!

How could such a process work in a traditional environment where a product owner or manager may not exist or has been be called away on another mission? How could it work if the developer is multiplexed across multiple projects or is working part-time "on assignment" from a resource pool? How could it work if the test resources are not dedicated and available at the same time that the code is written? The answer is, it doesn't.

Clearly, we are going to have refactor our organization to achieve this agile efficiency. We are going to have to *organize around the requirements* stream and build teams that have the full capability *to define, build, test, and accept* stories into the baseline—every day.

Eliminating the Functional Silos

Unfortunately, for many of us, we are not organized that way now. Instead, we are likely to be organized in functional silos, as illustrated in Figure 3–3.

Developers sit with, and communicate with, other developers. Product management, business analysts, and program managers are co-located with each other and often report to different departments entirely. For the larger organizations, architects may work together so as to help induce common architectures across business units, and so on, but may have little affinity or association with the development teams themselves. Product owners may not even exist, or if the function is filled by product managers, they are so multiplexed and/or unavailable that the team is constantly frozen, awaiting answers. Testers probably report to—and are co-located with—a separate QA organization, rather than development.

In agile, we must redefine what makes a team a team and eliminate the silos that separate the functions, as Figure 3–4 illustrates.

AGILE TEAM ROLES AND RESPONSIBILITIES

Fortunately, although it is no small matter to reorganize this way, the basic organizational structure of each agile team is largely the same, so at least we have a well-defined objective to achieve.

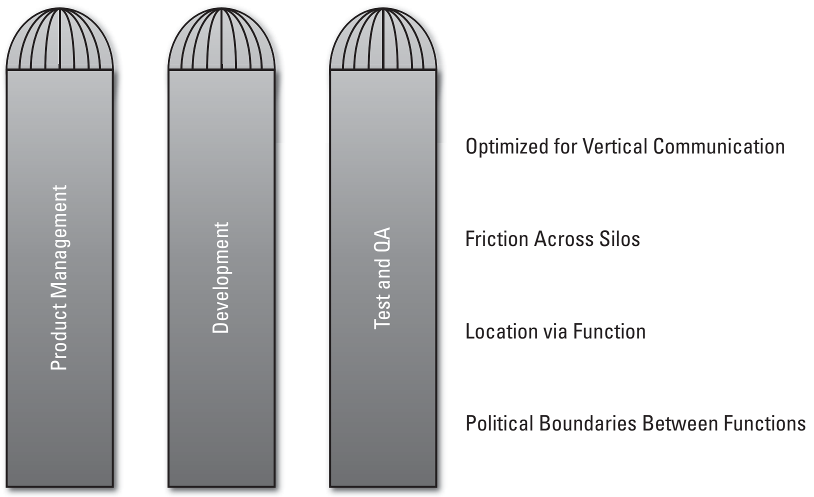

Figure 3–3 Typical functional silos

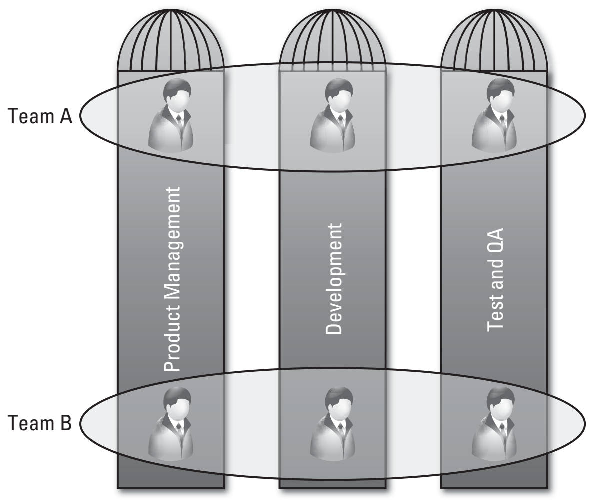

Figure 3–4 Reorganizing into agile teams

With Scrum, for example, there are three roles on an agile project team: the *product owner*, the *Scrum Master*, and the rest of the *team*, consisting primarily of the *developers* and *testers* who write and test the code.

Product Owner

Since the product owner is primarily responsible for defining and prioritizing requirements, it is clear why it is such a critical role in the agile project team. Chapter 11 is devoted to this topic.

In summary form, the product owner role is responsible for the following:

- Working with product managers, business analysts, customers, and other stakeholders to determine the requirements
- Maintaining the backlog and setting priorities based on relative user value
- Setting objectives for the iteration
- Elaborating stories, participating in progress reviews, and accepting new stories

Scrum Master/Agile Master

Scrum is quite specific about this role and provides specialized training and a specific title (Scrum Master) for those who assume this role. Although not every team will have a Scrum Master or apply Scrum per se, the method serves as a good example for all agile teams, and most agile teams have a Scrum Master or agile team lead, at least initially, who has some agile training and takes the initiative to fill this role. No matter the method, the Scrum/Agile Master is responsible for four things.

- *Facilitating the team's progress toward the goal:* Scrum/Agile Masters are trained as team facilitators and are constantly engaged in challenging the old norms of development while keeping the team focused on the goals of the iteration.
- *Leading the team's efforts in continuous improvement:* This includes helping the team improve, helping the team take responsibility for their actions, and helping the team become problem solvers for themselves.
- *Enforcing the rules of the agile process:* The rules of agile are lightweight and flexible, but they are rules nonetheless, and this role is responsible for reinforcing the rules with the team.
- *Eliminating impediments:* Many blocking issues will be beyond the team's authority or will require support from other teams. This role actively addresses these issues so that the team can remain focused on achieving the objectives of the iteration.

Developers

Developers write the code for the story. In so doing, they may work in a pair-programming model with another developer, they may be paired with a tester, or they may operate more independently and have interfaces to multiple testers and other developers. In any case, the responsibility is the same, and it includes the following:

- Collaborating with product owners and testers to make sure the right code is being developed
- Writing the code
- Writing and executing the unit test for the code
- Writing methods as necessary to support automated acceptance tests and other testing automation
- Checking new code into the shared repository every day

In addition, developers actively participate in improving the development environment.

Testers

Testers are an integral part of every agile team. They become part of the team just as soon as new code starts to be laid down, and they continue with the team throughout the release process. Their cycle is the same as the development cycle. Every new story that reaches an iteration boundary is subject to immediate review and analysis for acceptance testability. The tester's workflow in the course of the iteration parallels that of the developer:

- Writing the acceptance test case while the code is being written
- Interfacing with the developer and product owner to make sure the story is understood and that the acceptance tests track the desired functionality of the story
- Testing the code against the acceptance test
- Checking the test cases into the shared repository every day
- Developing ongoing test automation to integrate acceptance and component tests into the continuous testing environment

These are the primary roles on the agile team. At this level, that's all the understanding we will need to move forward with defining requirements practices for the team. In summary, Figure 3–5 illustrates an "ideal" agile team.

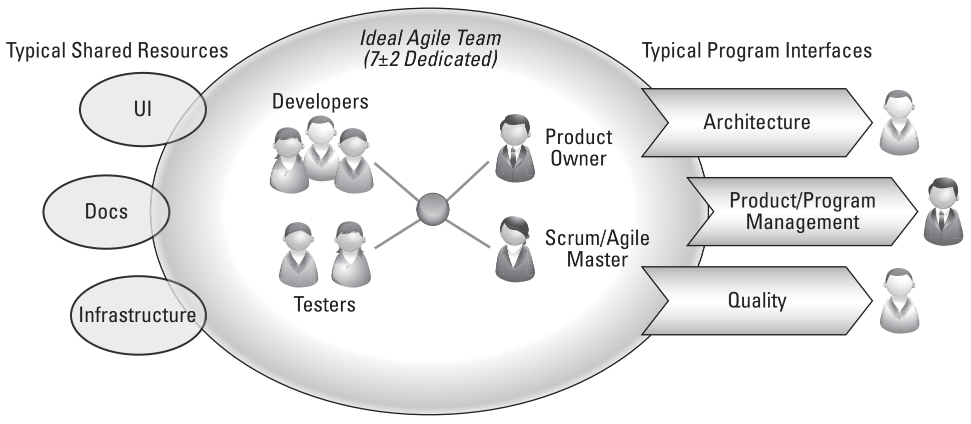

Figure 3–5 Ideal agile team with shared resources and typical interfaces

Other Team/Program Roles

You'll also note some shared resources and interfaces to other roles, including the following.

- *Architects:* Many agile teams do not contain people with titles containing the word *architect*,[1] and yet architecture does matter to agile teams. In these cases, the local architecture (that of the component, service, or feature that the team is accountable for) is most often determined by the local teams in a collaborative model. In this way, it can be said that "architecture emerges" from the activities of those teams.

  At the system level, however, architecture is often coordinated among system architects and business analysts who are responsible for determining the overall structure (components and services) of the system, as well as the system-level use cases and performance criteria that are to be imposed on the system as a whole. For this reason, it is likely that the agile team has a key interface to one or more architects who may live outside the team. (We'll discuss this in depth in Chapter 20.)

- *The role of quality assurance:* Quality assurance also plays a different role in agile. Since the primary responsibility for quality moves to the agile team (developers *and* testers), many QA personnel can typically reassume the role that was originally intended—that of overseeing overall, system-level quality by remaining "one step removed" from the daily team activities.

  Some of these QA personnel will live outside the team, while others (primarily testers) will have likely been dispatched to live with the product team. There, they work daily with developers to test new code and thereby help assure new code quality on a real-time basis.

  In addition, as we'll see later, QA personnel are involved with the development of the system-level testing required to assure overall system quality and conformance to nonfunctional, as well as functional, requirements.

- *Other specialists and supporting personnel:* Other supporting roles may include user-experience designers, documentation specialists, database designers and administrators, configuration management, build and deployment specialists, and whomever else is necessary to develop and deploy a whole product solution.

1. Agile Manifesto principle #11—*The best architectures, requirements, and designs emerge from self-organizing teams.*

User Stories and the Team Backlog

Since the efficiency of these agile teams is paramount to the overall organizational efficiency, we need to assure that the agile teams apply the simplest and leanest possible requirements model. To build a lean and scalable model, we need to make sure that the team's requirements artifacts are the simplest thing that could possibly support the needs of *all* stakeholders and particularly sensitive to the needs of the team members. Moreover, that subset must be quintessentially agile so that the artifacts described are consistent with most agile training as well as common practice. (In other words, it isn't mucked up with administrative overhead, manual traceability, reporting, detailed requirements cram down, or any of the other ways enterprises can unnecessarily burden the teams!)

Backlog

The term *backlog* was first introduced by Scrum, where it was described as a *product backlog*. However, in our enterprise model, *product* can be a pretty nebulous thing because various teams may be working at various levels, so there are multiple types of backlogs in the Big Picture. Therefore, our use of the term *backlog* is more generalized than in Scrum. In the Big Picture, we identified the particular backlog we are describing here as the team's local backlog, as shown in Figure 3–6.

This backlog is the one and only definitive source of work for the team. It holds all the work (primarily user stories) that needs to be done. It is local to them and is managed by them. It is their repository for all identified work items, and the contents are typically of little concern to others in the enterprise. They manage it, tool it, and put things in and out of it as it suits their needs in order to meet their iteration objectives. If "a thing to do" is in there, then it is likely to happen. If it isn't, then it won't.

Within the team, maintenance and prioritization of the backlog are the responsibility of the product owner, who is a resident of the team.

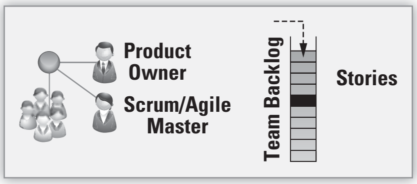

Figure 3–6 Stories and the team backlog

The team's backlog consists of all the work items the team has identified. In the meta-model, we generically call these work items *stories*[2] (some call them *backlogs* or *backlog items*) because that's what most agile teams and tools call them. For our purposes, we'll define a story simply as follows:

> A story is a work item contained in the team's backlog.

From a model perspective, a *story is a kind of backlog item*, as Figure 3–7 illustrates.[3]

User Stories

Although that definition is simple, it belies the underlying strength of agile in that it is a special kind of story, the *user story*, that agile teams use to define the system behavior and value for the user. Indeed, the user story is inseparable from agile's focus on value delivery. To make the user story explicit, we need to extend the model a little, as shown in Figure 3–8.

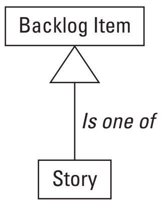

Figure 3–7 A story is a kind of backlog item.

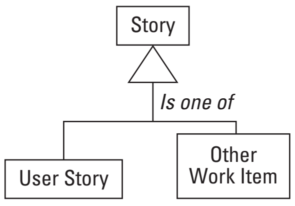

Figure 3–8 A story may be a user story or other work item.

2. Many Scrum teams call these *backlogs* because they are items in the product backlog. Strictly speaking, *work items* is probably a better term than story or backlogs, but we are trying to follow the most common usage as well as encouraging the use of the user story format.
3. The triangle indicator connecting the Story to Backlog item is the UML generalization relationship, indicating that the thing the arrow points to is a generalization of the special case of the pointing thing. In other words, in this case, story *is a special kind* of backlog item.

With this small addition, we now see that the backlog is composed of user stories and other work items. Other work items include things such as refactors, defects, support and maintenance, and tooling and infrastructure work. We'll discuss the rationale for specifically calling out these other work items later, but for now we just need to know that they help the team keep track of *all* the work they have to do to deliver value. They also help the team better estimate the time it will take to actually deliver the user stories.

User Story Basics

User stories are the agile replacement for most of what has been traditionally expressed as software requirements statements (or use cases in RUP and UML), and they are the workhorses of agile development. Developed initially within the constructs of XP, they are now endemic to agile development in general and are taught in most Scrum classes as well. We'll define a user story as follows:

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

As commonly taught, the user story takes a standard (user voice) form:

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

In this form, user stories can be seen to incorporate elements of the problem space (the business value delivered), the user's role (or persona), and the solution space (the activity that the user does with the system). Here's an example:

> "As a Salesperson (<role>), I want to paginate my leads when I send mass e-mails (<what I do with the system>) so that I can quickly select a large number of leads (<business value I receive>)."

User stories are so important that Chapter 6 is devoted entirely to this seminal agile construct.

Tasks

To assure that the teams really understand the work to be done and to assure that they can meet their commitments, many agile teams take a very detailed approach to estimating and coordinating the individual work activities necessary to complete a story. They do this via the *task*, which we'll represent as an additional model element, as is illustrated in Figure 3–9.

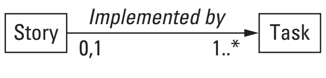

Figure 3–9 Stories are implemented by tasks.

Stories are implemented by tasks. Tasks are the lowest-granularity thing in the model and represent activities that must be performed by specific team members to accomplish the story. In our context:

> A task is a small unit of work that is necessary for the completion of a story.

Tasks have an owner (the person who has taken responsibility for the task) and are estimated in hours (typically four to eight). The burndown (completion) of task hours represents one form of iteration status. As implied by the one-to-many relationship expressed in the model, there is often more than one task necessary to deliver even a small story, and it's common to see a mini life cycle coded into the tasks of a story. Here's an example:

Story 51: Select photo for upload

Task 51.1: Define acceptance test—Juha, Don, Bill

Task 51.2: Code story—Juha

Task 51.3: Code acceptance test—Bill

Task 51.4: Get it to pass—Juha and Bill

Task 51.5: Document in user help—Cindy

In most cases, tasks are "children" to their associated story (deleting the story parent deletes the task). However, for flexibility, the model also supports stand-alone tasks and tasks that support other team objectives. With this construct, a team need not create a story simply to parent an item such as "install more memory in the file server."

ACCEPTANCE TESTS

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

- *Card* represents the two to three sentences used to describe the intent of the story.
- *Conversation* represents fleshing out the details of the intent of the card in a conversation with the customer or product owner. In other words, the card also represents a "promise for a conversation" about the intent.

4. *www.xprogramming.com/xpmag/expCardConversationConfirmation.htm*

- *Confirmation* represents how the team, via the customer or customer proxy, comes to understand that the code meets the full intent of the story.

▶ **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 agile project management tooling, but teams often still use physical cards for planning, estimating, prioritizing, and visibility in the daily stand-up.

With this simple alliteration and agile's zealousness for "all code is tested code," we have an object lesson in how quality is achieved during, rather than after, actual code development.

In our model, we represent the confirmation function as a type of *acceptance test*, one that confirms the story has been implemented correctly. To separate it from other types of acceptance tests (an overloaded term in software), we'll call them *story acceptance tests* and treat them as an artifact distinct from the (user) story itself, as shown in Figure 3–10.

There are many reasons why we did this, which we won't belabor here. In any case, the model is explicit in its insistence on the relationship between the story and the story acceptance test as follows.

- In the one-to-many (1..*) relationship, every story has one (or more) acceptance tests.
- It's *done* when it passes. A story cannot be considered complete until it has passed the acceptance test(s).

Acceptance tests are functional tests that verify that the system implements the story as intended. To avoid creating a large volume of manual tests, which would quickly limit the velocity of the team, story acceptance tests are automated wherever possible.

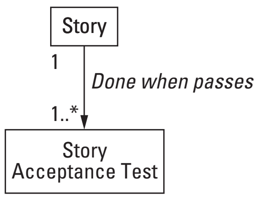

Figure 3–10 Every story has one or more story acceptance tests.

Unit Tests

To further assure quality, we can augment the acceptance with *unit tests*, as Figure 3–11 illustrates.

Unit tests are used to confirm that the lowest-level module of an application (a class or method in object-oriented programming; a function or procedure in procedural programming) works as intended. Unit tests are written by the developer to test that the code executes the logic of the subject module. In test-driven development (TDD), the test is written before the code. In any case, the test should be written, passed, and built into an automated testing framework before a story can be considered *done*.

Mature agile teams provide comprehensive practices for unit testing and automated functional (story acceptance) testing. Also, for those in the process of tooling their agile project, implementing this meta-model can provide inherent traceability of story-to-test, without any overhead on the part of the team.

Real Quality in Real Time

The combination of creating a lightweight story description, having a conversation about the story, elaborating the story into functional tests, augmenting the acceptance of the story with unit tests, and then automating testing is how agile teams achieve high quality in the course of each iteration. In this way:

> Quality is built in, one story at a time. Continued assurance of quality is achieved by continuous and automated execution of the aggregated functional and unit tests.

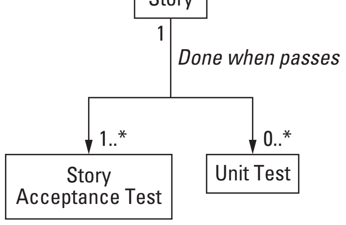

Figure 3–11 The code that implements the story should also be unit tested.

SUMMARY

In summary, we identified an organizational unit—the agile team—that eliminates functional silos and is optimized for the sole purpose of defining, building, and testing new functionality. We also described a set of requirements artifacts and relationships, including the user story, that are optimized to support the fast delivery of valuable requirements to the software baseline for release to the customers. We've also shown how agile teams achieve the highest-possible quality through comprehensive functional and unit testing and test automation.

In doing so, we've introduced a number of agile requirements artifacts. At the Team level, the requirements model isn't trivial, but it isn't that complex either, as Figure 3–12 summarizes.

In the next chapter, we'll move higher in the Big Picture and describe requirements practices for larger groups of teams, who work within agile *programs*. In later parts of this book, we'll describe how teams go about identifying, prioritizing, implementing, and delivering value using these artifacts.

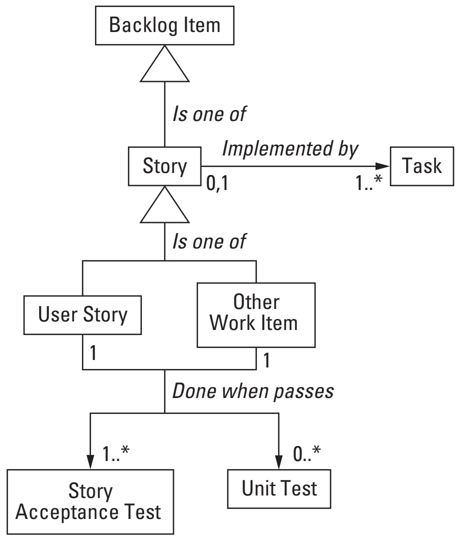

Figure 3–12 Agile team-level requirements artifacts and relationships

This page intentionally left blank