Chapter 19

Use Cases

A user story is to a use case as a gazelle is to a gazebo.

—Alistair Cockburn

In Chapter 1, we provided a brief history of requirements methods and briefly mentioned the role of use cases as a form of requirements capture and expression. Popularized originally within the context of the Rational Unified Process (RUP), which was *use case–driven* and architecture-centric, for many, use cases have been the requirements analysis and communication expression of choice. Even outside RUP, they appeared in most contemporary works on software requirements and systems analysis. Use cases were also the container for functional requirements capture, analysis, and behavioral specification within the context of the Unified Modeling Language (UML). Those who used the UML most likely used use cases. If you have been doing iterative development, you are probably using use cases too.

In agile development, however, the picture changed as the user story (or even more simply, the backlog item in Scrum) became the predominant form. As we have described throughout this book, there can be no doubt of their value in lightening requirements expression, driving more and more incremental thinking, and generally increasing the agility of the teams that use them. User stories are good requirements tools. Use cases were largely banned from the agile tribes.[1]

As agile methods started to be applied to larger systems, however, something was missing: *context*. Simply put, although a nice itemized list of backlog items is easy to look at, tool, prioritize, and manage, it is inadequate to do the more complex analysis work that larger systems require. And even though we've called our backlog items user stories, they don't really tell much of a story after all, at least not one much beyond what any casual reader might understand.

1. One Certified Scrum Product Owner course stated, "Don't use use cases. They are too hard to write, and users don't understand them."

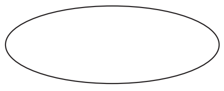
To this end, this chapter describes how to apply use cases in the development of complex software systems being developed in an agile manner. After all, who is building a simple software system with agile these days?

The Problems with User Stories and Backlog Items

Alistair Cockburn is one agile thought leader with his foot in both of these camps. As both an authority on use cases [Cockburn 2001] and a respected agilist and signer of the Agile Manifesto, he bemoans the apparent loss of use cases from agile development. He notes that there are many problems with the user story and backlog forms of requirements expression:[2]

> User stories and backlog items don't give the designers a context to work from—when is the user doing this, what is the context of their operation, and what is their larger goal at this moment?
>
> User stories and backlog items don't give the project team any sense of scope or potential "completeness"—a development team estimates a project at (e.g.) 270 story points, and then as soon as they start working, that number keeps increasing, seemingly without bound. The developers and sponsors are equally depressed. How big is this project, really?
>
> User stories and backlog items don't provide a mechanism for looking ahead at upcoming work. Seeing a set of extension conditions (alternate flows) in a use case lets the analysts understand which ones will be easy and which will be difficult so they can stage the work accordingly. With user stories, the extension conditions are usually detected mid-sprint, when it is too late.

Five Good Reasons to Still Use Use Cases



Use cases help explore the deeper interactions among *users*, the *systems*, and the *subsystems* of the solution. The use case also helps identify all the *alternate scenarios* that trip us up so often when it comes to system-level quality. This is especially the case when the team is building complex hardware and software systems, where system-spanning epics, features, and stories bob in and out of hardware and software like a surfacing porpoise. Cockburn notes "five good reasons to use use cases in agile development."

- *The list of goal names provides executives with a short summary of what the system will contribute to the business and the users. It also provides a project planning skeleton, to be used to build initial priorities, estimates, team allocation, and timing. It is the first part of the completeness question.*

2. See *http://alistair.cockburn.us/A+user+story+is+to+a+use+case+as+a+gazelle+is+to+a+gazebo*. Portions reproduced here with permission. Minor edits by the author.

- *The main success scenario of the use case provides everyone with an agreement as to what the system will . . . and will not do. It provides the context for each requirement, a context that is hard to get any other way.*
- *The extension conditions of the use case provide a framework for investigating all the little, niggling things that somehow take up 80% of the development time and budget. It provides a look-ahead mechanism, so the customer/product owner/ business analyst can spot issues that are likely to take a long time to get answers for. The use case extension conditions are the second part of the completeness question.*
- *The use case extension scenarios provide answers to the many detailed, tricky business questions programmers ask: "What are we supposed to do in this case?" (normally answered by, "I don't know, I've never thought about that case"). It is a thinking/documentation framework that matches the if . . . then . . . else statement that helps programmers think through such issues.*
- *The full use case set (use case model) shows that the developers/analysts have thought through every user's needs, every goal they have with respect to the system, and every business variant involved. This is the final part of the completeness question.*

Since use cases can play such a beneficial role in agile development, this chapter will provide a basic grounding in use cases so an agile team or enterprise can understand how to apply them. For additional background and depth on use cases, we refer you to other texts on the subject.[3]

USE CASE BASICS

We'll start with a definition:

> *A use case describes a sequence of actions between an actor and a system that produces a result of value for that actor.*

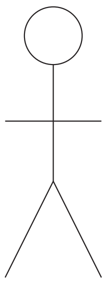
In other words, each use case describes a series of events in which a particular actor, such as Mark the *consumer*, interacts with a system, such as *the Tendril platform*, to achieve a result of value to Mark, such as shed some energy load in his area.

To further our understanding, let's look closer at this definition.

- *Sequence of actions:* The sequence of actions describes a set of interactions between the actor and the system. The sequence is invoked when the actor provides some input to the system. Each action is atomic; that is, it is performed either entirely or not at all.

3. There are dozens of books on the topic, including Cockburn's *Writing Effective Use Cases* [2001] and Leffingwell and Widrig's *Managing Software Requirements, Second Edition: A Use Case Approach* [2003].

- *System:* The system works for the actor. It executes some function, algorithmic procedure, or other activity. The system takes its orders from the actor as to when to do what.
- *A result of value:* Like a user story, the use case must deliver value to a user. Therefore, the resident pushes the opt-in button is not a good use case (the system didn't do anything obvious for the user). But the resident pushes the "opt-in" button and the system starts to shed load is a meaningful use case.
- *Actor:* The particular actor is the individual or device (Mark, the resident; a message from the utility) that initiates the action (shed some load; create a message on the user's TV).



Use Case Actors

An actor is someone or something that interacts with the system.

There are generally three types of actors to be considered.

- *Users:* Users act on the system; this is the type of actor most people think of when they think of a use case. For example, I am an actor on the word processing system I'm using to write this chapter.
- *Other systems or applications:* Most software interacts with other systems or other applications. This is another primary source of actors. Here's an example: The author's word processing application interacts with a Web service to access and insert clip art. The author's word processing application is an actor on the Web service.
- *A device:* Many applications interface to a variety of input and output devices. For example, the consumer's refrigerator is an actor on the Tendril platform.

Use Case Structure

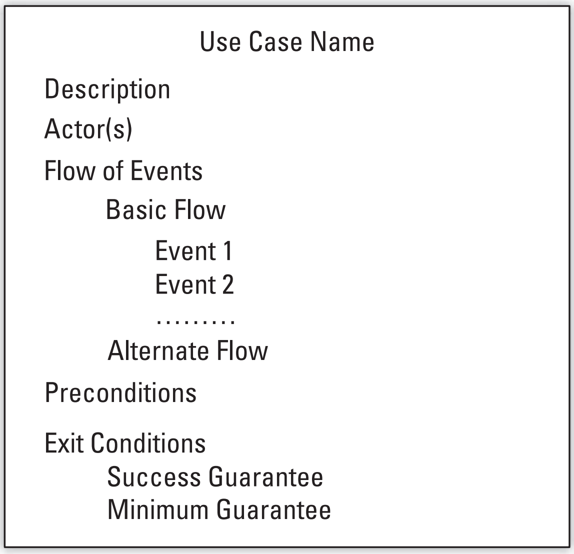
The use case itself is a text-based structure of logical elements that work together to define the use case. Figure 19–1 provides a standard template.

A use case has four mandatory elements.

- *Name:* The name describes the goal, that is, what is achieved by the interaction with the actor. The name should be a few words in length, and it must be unique. Names such as pop up an emergency warning message and shed refrigerator load are good examples. They are short, are descriptive, and define the goal.

- *Brief description:* The purpose of the use case should be described in one or two sentences. Here's an example: this use case describes what happens when the Tendril system receives an event notification from the utility.
- *Actor(s):* A use case has no meaning outside the context of its use by an actor, so each actor who participates in the use case is listed with the use case.
- *Flow of events:* The main body of the use case is the event flow, usually a textual description of the interactions between the actor and the system. The flow of events can consist of both the *basic flow*, which is the main path through the use case, and *alternate flows*, which are executed only under certain circumstances.

  It is the *alternate flows* (or *alternate scenarios*) that provide much of value to the agile system builder. It is here where they are forced to think through all the "what ifs" that might affect our user story. Here's an example: "what happens if the device does not respond to the message?" or "if I'm programming the system, and an event happens—what happens then?" Understanding all the alternate flows of a use case defines the various (usually less likely but equally important) scenarios that the system must handle with grace in order to assure reliability and quality.



```
                 Use Case Name
Description
Actor(s)
Flow of Events
        Basic Flow
              Event 1
              Event 2
              .........
        Alternate Flow
Preconditions
Exit Conditions
        Success Guarantee
        Minimum Guarantee
```

Figure 19–1 Use case template

In addition to the mandatory elements, a use case may have optional elements.

- *Preconditions:* Preconditions are those conditions that must be present in order for a use case to start. They usually represent some system state that must be present before the use case can be used. For example, a precondition of the *set thermostat to shed load* use case is that the system must have opt-in for load shedding enabled.
- *Exit conditions:* Exit conditions describe the state of the system after a use case has run its course. These can include both a *success guarantee*, which describes the state of the system after a successful execution, and a *minimum guarantee*, which describes the state of the system if the execution of the use case fails for some reason. For example, a *success guarantee* of the *set thermostat to shed load* use case is that the system remains in the load shedding state. A *minimum guarantee* might be that load shedding is not initiated but an error message is displayed.
- *System or subsystem:* In a system of subsystems, it may be necessary to identify whether a use case is a system-level use case (one that causes multiple subsystems to interact) or a subsystem use case. In either case, the team needs to identify what system or subsystem a use case is identified with.
- *Other stakeholders:* It may also be useful to identify other key stakeholders who may be affected by the use case. For example, a manager may use a report from the system, and yet the manager may not personally interact with the system and would therefore not appear as an actor.
- *Special requirements:* The use case may also have special requirements that apply to that specific use case. Often these are nonfunctional requirements (*support 10,000 homes without performance degradation*) that apply to the specific use case.

A Step-by-Step Guide to Building the Use Case Model

An individual use case describes how a particular actor interacts with the system to achieve a result of value for that specific actor. The set of all use cases together describes the complete behavior of the system. The complete set of use cases, actors, and their interactions constitutes the *use case model* for the system.

Building the use case model for the system is an important analysis step, one that will become the basis for understanding, communicating, and refining the behavior of the system over the course of the project. We have found a simple five-step approach to be an effective way to build the use case model. However, these steps do not all happen at the same point in the project life cycle, and some steps will likely be revisited.

Step 1: Identify and Describe the Actors

First, identify all the actors that interact with the system. This is a matter of dividing the world into two classes of interesting things: the system we are building and those things (actors) that interact with the system.

The following questions will help identify actors.

- Who uses the system?
- Who gets information from this system?
- Who provides information to the system?
- Where is the system used?
- Who supports and maintains the system?
- What other systems or devices use this system?

Step 2: Identify the Use Cases

Once the actors are identified, the next step is to identify the various use cases that the actors need to accomplish their jobs. We can do this by determining the specific goals for each actor in turn.

- What will the actor use the system for?
- Will the actor create, store, change, remove, or read data in the system?
- Will the actor need to inform the system about external events or changes?
- Will the actor need to be informed about certain occurrences in the system?

We might discover use cases such as *send consumers a notification of pending event* as something that a power utility's network operations center (the actor) might want to do to the Tendril platform. Graphically, we could construct a simple diagram of use cases and actors as illustrated in Figure 19–2.

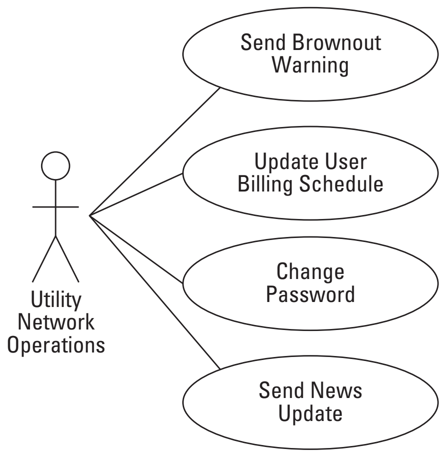


Figure 19–2 Use cases for the utility network operations center

Each use case has a *name* that represents the goal of the use case. The name is a few words or short phrase that starts with an action verb and communicates what the actor achieves. Names help communicate what the system does and create context.

Along with the name, there is a brief description that further describes the intent of the use case.

Step 3: Identify the Actor and Use Case Relationships

Although only one actor can initiate a use case, some use cases involve the participation of multiple actors. When the actors and use cases interact in concert, that's when the system becomes a system. In this step, each use case is analyzed to see what actors interact with it, and each actor's anticipated behavior is reviewed to verify that the actor participates in all the necessary use cases.

Step 4: Outline the Flow of the Use Cases

The next step is to outline the flow of each use case to start to gain an understanding of required system behavior at the next level of detail. Of particular interest at this time is the flow of events, including the basic and alternate flows, as Figure 19–3 illustrates.

Outline the basic flow first. This is the flow that represents the most common path (the "happy path") from start to finish through the use case.

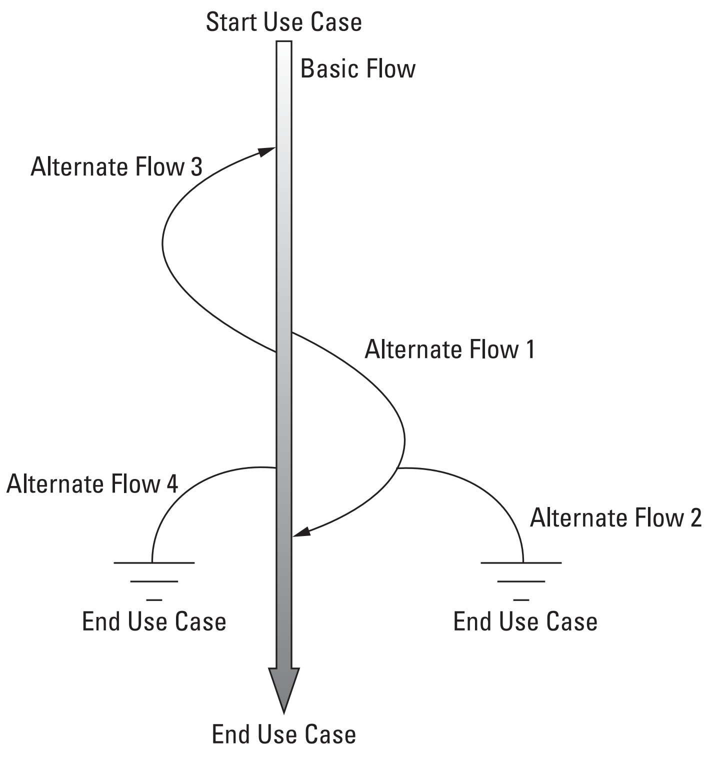


Figure 19–3 A use case has one main flow and some number of alternate flows.

In addition to the basic flow, the use case will have a number of alternate flows based on both regular circumstances and exceptional events. The following questions can help discover these paths.

- Basic flow
  - What actor's event starts the use case?
  - How does the use case end?
  - How does the use case repeat some behavior?
- Alternate flow
  - What else can the actor do?
  - How will the actor react to optional situations?
  - What variants might happen?
  - What exceptions to the usual behavior may occur?

Step 5: Refine the Use Cases

Later, the use case may be refined to another level of detail, or perhaps the team will move directly to the stories that implement the use case. There are a number of additional factors to be considered.

- *Consider all alternate flows, including unusual exception conditions:* It is usually straightforward to identify the primary alternate flows of a use case since these are driven by explicit actor choices. However, the "what ifs" are a primary source of concern, and these must be fully explored in alternate flows. "What if the resident is programming the system when an energy event occurs?" All these exceptions must be understood, coded, and tested, or the application may not behave as expected.
- *Preconditions:* The refinement process will identify state information that controls the behavior of the system. Preconditions describe the things that must be true before a use case starts. For example, a precondition to programming vacation settings might be that the user has set the time zone. If that has not been done, the use case cannot be successfully executed.
- *Exit conditions:* These describe the persistent states the use case leaves behind. They can include the success guarantee (state after successful execution) and the minimum guarantee (state after unsuccessful execution).

A USE CASE EXAMPLE

As an example, we'll describe a use case from the Tendril platform. One of the system features is the capability of the power utilities to notify their consumers of pending changes in power distribution. Suppose a utility is about to go into a "brownout." The utility would like to notify its consumers of the impending brownout event so they could plan accordingly. Figure 19–4 illustrates a use case for this.

Use Case Name: Issue brownout notification
Brief Description: Upon determining that a brownout condition is pending, the utility sends a message to all registered devices in the utility's Tendril domain. This will include notification to all the utility's registered on-premise displays, mobile devices, and portals.
Actors:
• Utility Network Operating Center operators (UNOC) (primary actor)
• Tendril customers
Basic Flow of Events:
1. A UNOC operator determines that a brownout event is pending.
2. The operator composes a broadcast message in TUP.
3. TUP sends the message to all affected customers, either over the utility backhaul or through an Internet (IP) connection, to all registered Tendril devices in the consumer's home, and logs that the message has been sent.
4. Each Tendril device in the home returns to TUP a confirmation of successful message receipt and presents the upcoming brownout message to the homeowner in its device-specific format.
5. TUP adds each confirmation to its record of confirmations received.
Alternate Flows:
• 4a. Home Tendril device fails to confirm (initially): At established intervals, TUP resends the brownout event notice to all Tendril devices that did not reply.
• 4b. Home Tendril device fails to confirm after the specified number of retries: TUP notifies the UNOC operator of the situation of the nonacknowledging Tendril home device.
Preconditions:
• Receipt of overload conditions pending on the utility grid. • Determination of areas to be affected by brownout and matching areas to preset Tendril notification zones.
Success Guarantee:
• The brownout notice has been sent. • Every home device has been accounted for, through either confirmation or total failure to confirm. • The UNOC operator as been notified of all home devices that failed to confirm.
Minimal Guarantee:
• In the worst case, the TUP log has captured the state of all notification-confirmation pairs for every home device on the subscription list.

Figure 19–4 A sample use case

We can see from the example that a use case is not a trivial thing. Rather, it is an appropriate requirements artifact used to capture and define the behavior of non-trivial systems. One could imagine that a single use case like the one in Figure 19–4 could create dozens, or more, individual user stories. But the use case spawns these stories within a system usage context. That can be a tremendous aid to understanding for the customer, user, development team, and other key stakeholders alike.

APPLYING USE CASES

If your system is complex and is composed of subsystems (what complex system isn't?), use cases provide a mechanism to help developers think about all the possible paths through the system and subsystems that the users may encounter.

In turn, this helps the team understand where user stories are likely to be needed to implement the required functionality. Every time a new use case touches a subsystem, there are probably new stories that must be developed for that subsystem, as is shown in Figure 19–5.

Once identified, the use cases can then be used to drive incremental development, one story at a time, as Kroll and MacIsaac [2006] illustrate in Figure 19–6.

In this graphic, one can see how use case–spawned user stories can be implemented in iterations over time (story 1 in iteration 1, stories 2 and 3 in iteration 2, and so on). To the agile developer or product owner, this can be helpful in understanding the larger context that can be lost when the teams focus on understanding "just one user story at a time."

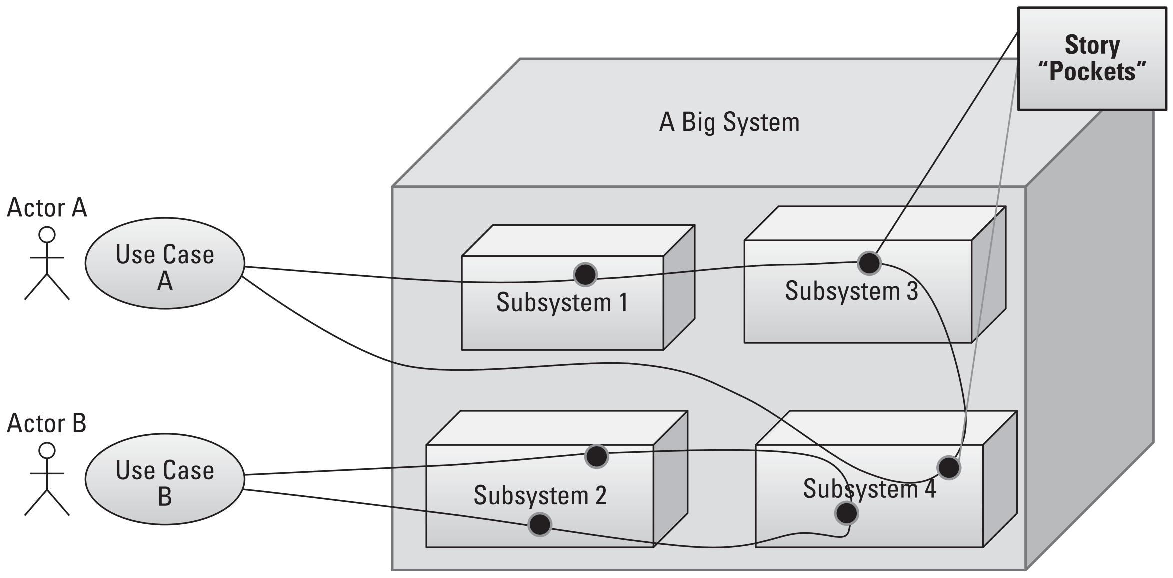


Figure 19–5 Use cases traversing a system identify where stories are needed.

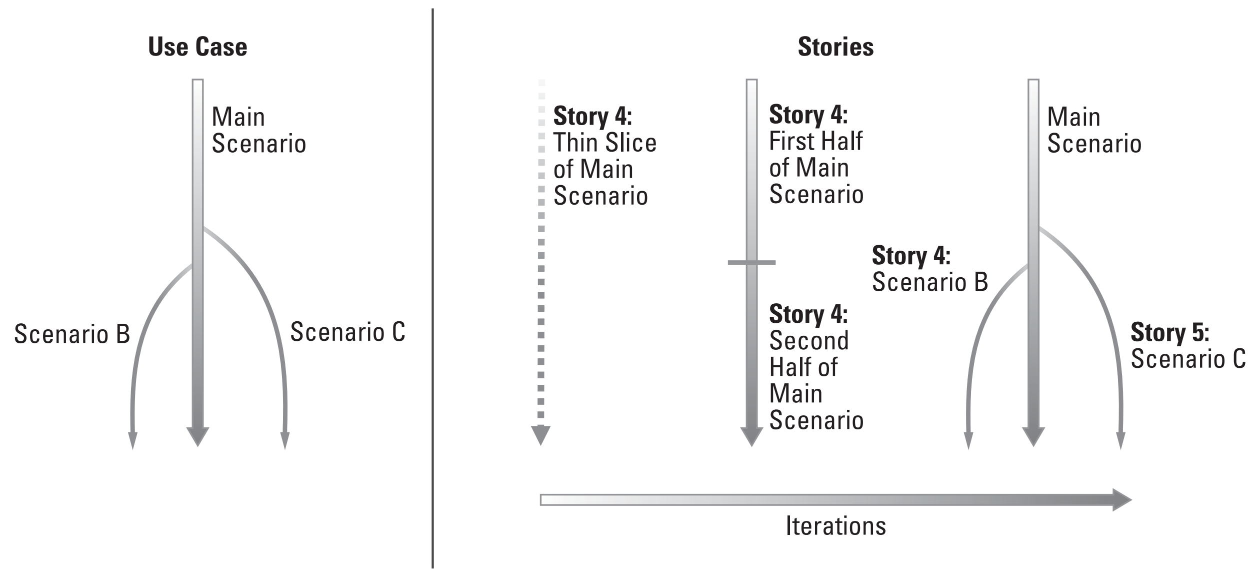

Use Case
Main Scenario
Scenario B
Scenario C
Stories
Story 4: Thin Slice of Main Scenario
Story 4: First Half of Main Scenario
Story 4: Second Half of Main Scenario
Story 4: Scenario B
Main Scenario
Story 5: Scenario C
Iterations

Figure 19–6 Sequencing user stories in iterations
Source: Kroll/MacIsaac, *Agility and Discipline Made Easy*, © 2006

Tips for Applying Use Cases in Agile

Finally, here are a few tips to keep in mind when you apply use cases in agile development.

- Keep them lightweight—no design details, GUI specs, and so on.
- Don't treat them like fixed requirements. Like user stories, they are merely statements of intended system behavior.
- Don't worry about maintaining them; they are primarily thinking tools.
- Model them informally—use whiteboards, lightweight tools, and so on.

Remember, you don't have to use use cases, but no one can tell you not to use them either. And if your system is complex, you'll likely be quite happy if you do.

USE CASES IN THE AGILE REQUIREMENTS INFORMATION MODEL

As we have described, use cases are an *optional* technique, which teams can use to better understand desired system behaviors. We haven't yet introduced them in the agile requirements meta-model.

Since functional system behaviors are captured as backlog items, we'll associate use cases there, as an optional requirements modeling and elaboration artifact. We illustrate this in Figure 19–7.

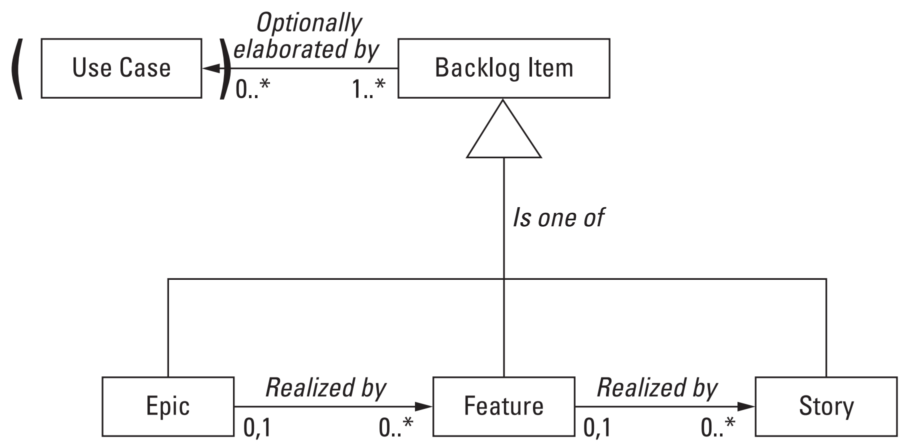


Figure 19–7 Use cases may be used to elaborate desired system behavior.

As the model indicates, you can apply use cases to describe more complex system behavior, most typically at the feature and epic levels, where they serve to better illustrate *what we mean by that big thing*.

SUMMARY

Throughout most of this book, we've used user stories, features, and epics to define the behavior of the systems we are building. They work well, and they have earned their place in the forefront of agilists' minds. As convenient as they are, however, they can do a poor job of helping teams define and understand the behavior of more complex systems, where we need to understand the user and the system *in context*, not to mention thinking through all the "what if" scenarios that happen with systems of any scale. For this we can apply use cases, which are a well-understood and proven tool that was invented for just this purpose.

Even though use cases aren't generally promoted in agile, as professionals we need to keep an open mind about these things. Ignoring techniques that have helped us manage complexity before is stupid. And being stupid isn't lean, agile, professional, or economically sound.