

Chapter 11

When Use Cases Aren't Enough

In this chapter:	
The Power of Use Cases	89
Project Type Limitations	90
Event-Response Tables	91
Use Cases Don't Replace Functional Requirements	93
Use Cases Reveal Functional Requirements	96

Use cases are recognized as a powerful technique for exploring user requirements. The great benefit they provide is to bring a user-centric and usage-centered perspective to requirements elicitation discussions. The analyst employs use cases to understand what the user is trying to accomplish and how he envisions his interactions with the product leading to the intended user value. Putting the user at the center is much better than focusing on product features, menus, screens, and functions that characterize traditional requirements discussions. And the structure that use cases provide is far superior to the nearly worthless technique of asking users, "What do you want?" or "What are your requirements?"

As with most new software development techniques, use cases have acquired a bit of a mystique, some misconceptions, overblown hype, and polarized camps of enthusiasts who will all try to teach you the One True Use Case Way. In this chapter, I share my perspectives on when use cases work well, when they don't, and what to do when use cases aren't a sufficient solution to the requirements problem.

The Power of Use Cases

I'm a strong believer in the use case approach. Use cases are an excellent way to structure the dialogue with users about the goals they need to accomplish with the help of the system. Users can relate to and review use cases because the analyst writes them from the user's point of view, describing aspects of the user's business. In my experience, once they get past the discomfort of trying a new technique, users readily accept the use case method as a way to explore requirements.

I'm often asked how to write requirements specifications so that users can read and understand them and also so that they contain all the detail that developers need. In many cases, one requirements representation won't meet both of these objectives. Users can comprehend use cases, but they might balk at reviewing a more detailed functional requirements specification. Use cases give developers an overall understanding of the system's behavior

that fragments of individual functionality cannot. However, developers usually need considerably more information than use cases provide so that they know just what to build. In many circumstances, the combination of employing use cases to represent user requirements and a software requirements specification to contain functional and nonfunctional requirements meets both sets of needs.

Project Type Limitations

My experience has shown that use cases are an effective technique for many, but not all, types of projects. Use cases focus on the user's interactions with the system to achieve a valuable outcome. Therefore, use cases work great for interactive end-user applications, including Web sites. They're also useful for kiosks and other types of devices with which users interact.

However, use cases are less valuable for projects involving data warehouses, batch processes, hardware products with embedded control software, and computationally intensive applications. In these sorts of systems, the deep complexity doesn't lie in the user-system interactions. It might be worthwhile to identify use cases for such a product, but use case analysis will fall short as a technique for defining all the system's behavior.



I once worked on a computational program that modeled the behavior of a multi-stage photographic system. This software used a Monte Carlo statistical simulation to perform many complex calculations and it presented the results graphically to the user. The user-system dialog needed to set up each simulation run was quite simple. (I know this because I built the user interface.) The complexity resided behind the scenes, in the computational algorithms used and the reporting of results. Use cases aren't very helpful for eliciting the requirements for these aspects of a system.

Use cases have limitations for systems that involve complex business rules to make decisions or perform calculations. Consider an airline flight reservation system, one of the classic examples used to illustrate use cases. Use cases are a fine technique for exploring the interactions between the traveler and the reservation system to describe the intended journey and the parameters associated with it. But when it comes to calculating the fare for a specific flight itinerary, a use case discussion won't help. Such calculations are driven by the interaction of highly complex business rules, not by how the user imagines conversing with the system.

Nor are use cases the best technique for understanding certain real-time systems that involve both hardware and software components. Think about a complex highway intersection. It includes sensors in the road to detect cars, traffic signals, buttons pedestrians can press to cross the street, pedestrian walk signals, and so forth. Use cases don't make much sense for specifying a system of this nature. Here are some possible use cases for a highway intersection:

- A driver wants to go through the intersection.
- A driver wants to turn left when coming from a particular direction.
- A pedestrian wants to cross one of the streets.

These approximate use cases, but they aren't very illuminating. Exploring the interactions between these actors (drivers and pedestrians) and the intersection-control software system just scratches the surface of what's happening with the intersection. The use cases don't provide nearly enough information for the analyst to define the functionality for the intersection-control software.

Use cases aren't particularly helpful for specifying the requirements for batch processes or time-triggered functions, either. My local public library's information system automatically sends me an e-mail to remind me when an item I've borrowed is due back soon. This e-mail is generated by a scheduled process that checks the status of borrowed items during the night (the one I received today was sent at 1:06 AM) and sends out notifications. Some analysts regard "time" to be an actor so that they can structure this system behavior in the form of a use case. I don't find that helpful, though. If you know the system needs to perform a time-triggered function, just write the functional requirements for that function, instead of packaging it into a contrived use case.

Event-Response Tables

A more effective technique for identifying requirements for certain types of systems is to consider the external events the system must detect. Depending on the state of the system at the time a given event is detected, the system produces a particular response. Event-response tables are a convenient way to collect this information (Wiegiers 2003a). Events could be signals received from sensors, time-based triggers (such as scheduled program executions), or user actions that cause the system to respond in some way. Event-response tables are related to use cases. In fact, the trigger that initiates a use case is sometimes termed a *business event*.

The highway intersection system described earlier has to deal with various events, including these:

- A sensor detects a car approaching in one of the through lanes.
- A sensor detects a car approaching in a left-turn lane.
- A pedestrian presses a button to request to cross a street.
- One of many timers counts down to zero.

Exactly what happens in response to an external event depends on the state of the system at the time it detects the event. The system might initiate a timer to prepare to change a light from green to amber and then to red. The system might activate a Walk sign for a pedestrian (if the sign currently reads Don't Walk), or change it to a flashing Don't Walk (if the sign currently says Walk), or change it to a solid Don't Walk (if it's currently flashing). The analyst needs to write the functional requirements to specify ways to detect the events and the decision logic involved in combining events with states to produce system behaviors. Table 11-1 presents a fragment of what an event-response table might look like for such a system. Each expected system behavior consists of a combination of event, system state, and response.

press the accelerator and the brake pedal, turn the steering wheel, and shift gears, but these aren't truly use cases—they're events. A great deal of the product's complexity lies not in the user interactions but under the hood (literally, in this case). An event-response approach will go much farther toward understanding the requirements for this kind of system. So although use cases are valuable for systems in which much of the complexity lies in the interactions between the user and the computer, they are not effective for some other types of products.

Use Cases Don't Replace Functional Requirements

One book about use cases states, "To sum up, all functional requirements can be captured as use cases, and many of the nonfunctional requirements can be associated with use cases" (Bittner and Spence 2003). I agree with the second part of this sentence but not with the first part. It is certainly true that use cases are a powerful technique for discovering the functional requirements for a system being developed. However, this statement suggests that use cases are the only tool needed for representing a software system's functionality.

This notion that all functional requirements can fit into a set of use cases and that use cases contain all the functional requirements for a system appears in many of the books and methodologies that deal with use cases. The thinking seems to be that the use cases *are* the functional requirements. If the analyst writes good use cases, the developers are supposed to be able to build the system from that information, along with nonfunctional requirements information that's included in a supplementary specification.¹ Nonfunctional requirements, such as performance, usability, security, and availability goals, typically relate to a specific use case or even to a particular flow within a use case.

Unfortunately, despite the thousands of students I've taught in requirements seminars over the years, I have yet to meet a single person who has found this pure use case approach to work! Perhaps some people have successfully done it, but I haven't met any of them. On the contrary, dozens of requirements analysts have told me, "We gave our use cases to the developers and they got the general idea, but the use cases just didn't contain enough information. The developers had to keep asking questions to get the additional requirements that weren't in the use cases." I suppose you could argue that they must not have been very good use cases. But when dozens of people report the same unsatisfactory experience when trying to apply a particular methodology, I question the methodology's practicality.

¹ There's an interesting conflict in the current use case literature. Bittner and Spence (2003) provide the following definition for *supplementary requirements*: "Functional or nonfunctional requirements that are traceable to a particular use case are said to *supplement* the use case description" (their italics). However, the Unified Software Development Process, which is heavily use case driven, offers a definition of *supplementary requirement* that is directly contradictory: "A generic requirement that cannot be connected to a particular use case..." (Jacobson, Booch, and Rumbaugh 1999). It's no wonder practitioners get confused. It's generally agreed that a supplemental specification is needed to contain at the very least those nonfunctional requirements that the use cases do not describe.

94 Part IV: On Use Cases

There are three problems with adhering to this philosophy of use case purity. First, your use cases must contain all the functional detail that the analysts need to convey to the developers. That requires writing highly detailed use cases. The sample use cases in some books do include some complex examples. But elaborate use cases become hard to read, review, and understand.

The second problem with this approach is that it forces you to invent use cases to hold all the functional requirements because a use case is the only container you have available to describe system functionality. However, some system functionality does not fit appropriately into a use case. I have seen many new use case practitioners struggle to create inappropriate use cases to hold all the bits of functionality, to no useful end.

Logging in to an ATM or a Web site is an example that illustrates this problem. Bittner and Spence (2003) provide this good definition of use case:

Describes how an actor uses a system to achieve a goal and what the system does for the actor to achieve that goal. It tells the story of how the system and its actors collaborate to deliver something of value for at least one of the actors.

By this definition, logging in to a system is not a legitimate use case because it provides no value to the person who is logging in. No one logs in to a system and feels as though he accomplished something as a result. Logging in is a means to an end, a necessary step to being able to perform use cases that do provide value. Nevertheless, the functionality to permit login and everything associated with it (such as business rules or integrity requirements regarding passwords) must be defined somewhere. If you're using only use cases to capture functional requirements, you wind up inventing artificial use cases—those that don't provide user value—just to have a place to store certain chunks of functionality. This artificiality does not add value to the requirements development process.

A third shortcoming of the use case-only philosophy is that use cases are organized in a way that makes good sense to users but not to developers. As Figure 11-1 illustrates, a use case consists of multiple small packages of information. A typical use case template contains sections for preconditions, postconditions, the normal flow of events, zero or more alternative flows (labeled with *Alt.* in Figure 11-1), zero or more exceptions (labeled with *Ex.*), possibly some business rules, and perhaps some additional special requirements.

These small packages are easy to understand and review, but they make it hard for the developer to see how the pieces fit together. As a developer, I find it more informative to see all the related requirements grouped together in a logical sequence. Suppose I read a functional requirement that implements a step in the normal flow. I want to see the requirements dealing with branch points into alternative flows and conditions that could trigger exceptions immediately following that one functional requirement. I'd like to see the requirements that handle each business rule in context, juxtaposed with the relevant system functionality.

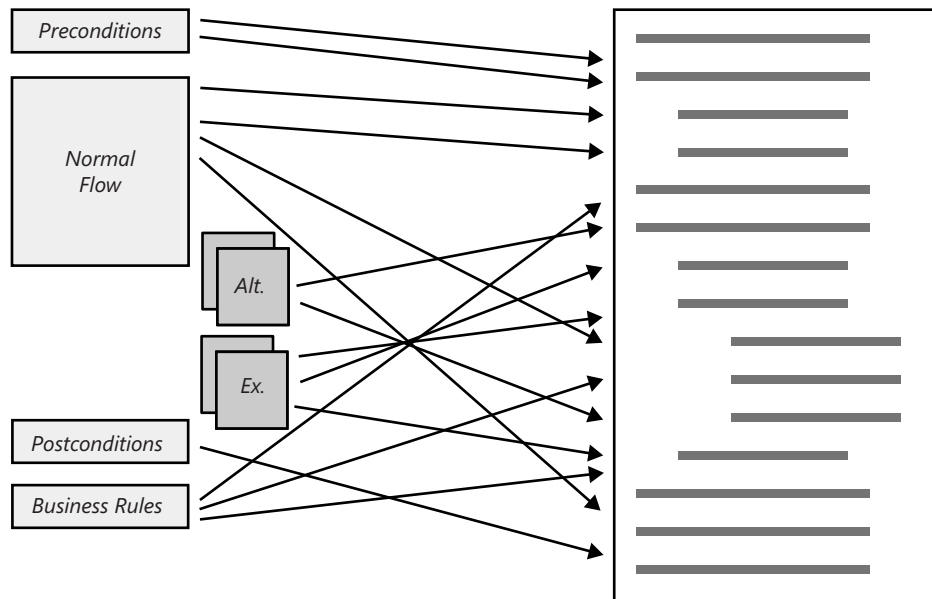


Figure 11-1 Use case organization (left) differs from SRS organization (right).

As Figure 11-1 illustrates, the functional requirements that come from the various chunks of a use case can be sprinkled throughout a hierarchically organized SRS. Traceability analysis becomes important so that you can make sure every functional requirement associated with the use case traces back to a specific part of the use case. You also want to ensure that every piece of information in the use case leads to the necessary functionality in the SRS. In short, the way a use case is organized is different from the way many developers prefer to work.

It gets even more confusing if you're employing use cases to describe the bulk of the functionality but have placed additional functional requirements that don't relate to specific use cases into a supplemental specification (Leffingwell and Widrig 2003). This approach forces the developer to get some information from the use case documentation and then to scour the supplemental specification for other relevant inputs. Before your analysts impose a particular requirements-packaging strategy on the developers, have these two groups work together to determine the most effective ways to communicate requirements information. (See Chapter 12, "Bridging Documents.")

My preference is for the analyst to create an SRS as the ultimate deliverable for the developers and testers. This SRS should contain all the known functional and nonfunctional requirements, regardless of whether they came from use cases or other sources. Functional requirements that originated in use cases should be traced back to those use cases so that readers and analysts know where they came from.

Use Cases Reveal Functional Requirements

Rather than expecting use cases to contain 100 percent of the system's functionality, I prefer to employ use cases to help the analyst discover the functional requirements. That is, the use cases become a tool rather than being an end unto themselves. Users can review the use cases to validate whether a system that implemented the use cases would meet their needs. The analyst can study each use case and derive the functional requirements the developer must implement to realize the use case in software. I like to store those functional requirements in a traditional SRS, although you could add them to the use case description if you prefer.

I'm often asked, "Which comes first: use cases or functional requirements?" The answer is use cases. Use cases represent requirements at a higher level of abstraction than do the detailed functional requirements. I like to focus initially on understanding the user's goals so that we can see how they might use the product to achieve those goals. From that information, the analyst can derive the necessary functionality that must be implemented so that the users can perform those use cases and achieve their goals.

Functional requirements—or hints about them—lurk in various parts of the use case. The remainder of this chapter describes a thought process an analyst can go through to identify the less obvious functional requirements from the elements of a use case description.

Preconditions

Preconditions state conditions that must be true before the actor can perform the use case. The system must test whether each precondition is true. However, use case descriptions rarely state what the system should do if a precondition is *not* satisfied. The analyst needs to determine how best to handle these situations.

Suppose a precondition for one use case states, "The patron's account must be set up for payroll deduction." How does the system behave if the patron attempts to perform this use case but his account is not yet set up for payroll deduction? Should the system notify the patron that he can't proceed? Or should the system perhaps give the patron the opportunity to register for payroll deduction and then proceed with the use case? Someone has to answer these questions and the SRS is the place to provide the answers.

Postconditions

Postconditions describe outcomes that are true at the successful conclusion of the use case. The steps in the normal flow naturally lead to certain postconditions that indicate the user's goal has been achieved. Other conditions, however, might not be visible to the user and therefore might not become a part of a user-centric use case description.

Consider an automated teller machine. After a cash withdrawal, the ATM software needs to update its record of the amount of cash remaining in the machine by subtracting the amount withdrawn. Perhaps if the cash remaining drops below a predetermined threshold the system

is supposed to notify someone in the bank to reload the machine with additional money. I doubt any user will ever convey this information during a discussion of user requirements, yet the developer needs to know about this functionality.

How can you best communicate this knowledge to the developers and testers? There are two options. One is to leave the use case at the higher level of abstraction that represents the user's view and have the requirements analyst derive the additional requirements through analysis. The analyst can place those requirements in an SRS that is organized to best meet the developer's needs. The second alternative is for the analyst to include those additional details directly in the use case description. That behind-the-scenes information is not part of the user's view of the system as a black box. Instead, you can think of that information as being white-box details about the internal workings of the use case that the analyst must convey to the developer.

Normal and Alternative Flows

The functionality needed to carry out the dialogue between the actor and the system is usually straightforward. Simply reiterating these steps in the form of functional requirements doesn't add knowledge, although it might help organize the information more usefully for the developer. The analyst needs to look carefully at the normal flow and alternative flows to see if there's any additional functionality that isn't explicitly stated in the use case description. For example, under what conditions should the system offer the user the option to branch down an alternative flow path? Also, does the system need to do anything to reset itself so that it's ready for the next transaction after the normal flow or an alternative flow is fully executed?

Exceptions

The analyst needs to determine how the system could detect each exception and what it should do in each case. A recovery option might exist, such as asking the user to correct an erroneous data entry. If recovery isn't possible, the system might need to restore itself to the state that existed prior to beginning the use case and log the error. The analyst needs to identify the functionality associated with such recovery and restore activities and communicate that information to the developer.

Business Rules

Many use cases are influenced by business rules. The use case description should indicate which business rules pertain. It's up to the analyst to determine exactly what functionality the developer must implement to comply with each rule or to enforce it. These derived functional requirements should be recorded somewhere (I recommend documenting them in the SRS), rather than just expecting every developer to figure out the right way to comply with the pertinent business rules.

Special Requirements and Assumptions

The use case might assume that, for instance, the product's connection to the Internet is working. But what if it's not? The developer must implement some functionality to test for this error condition and handle it in an appropriate way.

In my experience, the process of having the analyst examine a use case in this fashion to derive pertinent functional requirements adds considerable value to the requirements development process. There's always more functionality hinted at in the use case than is obvious from simply reading it. Someone needs to discern this latent functionality. I would prefer to have an analyst do it rather than a developer. If your developers are sufficiently skilled at requirements analysis they could carry out this task, but they might not view it as part of their responsibilities.



I recently spoke to a highly experienced developer who said it was much more helpful to receive requirements information organized in a structured way from the analyst than to have to figure out those details on his own. This developer preferred to rely on the analyst's greater experience with understanding the problem domain and deriving the pertinent functional requirements. Not only did this result in better requirements, but it also allowed the developer to focus his talents and energy where he added the most value—in designing and coding the software.

My philosophy of employing use cases as a tool to help me discover functional requirements means that I don't feel a need to force every bit of functionality into a use case. It also gives me the option of writing use cases at whatever level of detail is appropriate. I might write some use cases in considerable detail to elaborate all their alternative flows, exceptions, and special requirements. I could leave other use cases at a high level, containing just enough information for me to deduce the pertinent functional requirements on my own. That is, I view use cases as a means to an end. The functional requirements are that "end," regardless of where you choose to store them or whether you even write them down at all.

Deriving functional requirements from the use case always takes place on the journey from ideas to executable software. The question is simply a matter of who you want to have doing that derivation and when. (See Chapter 13, "How Much Detail Do You Need?") If you deliver only use cases without all the functional detail to developers, each developer must derive the additional functionality on his own. A developer with little requirements analysis expertise might overlook some of these requirements. It's also unlikely that all developers will record the functional requirements they identify. This makes it hard for testers to know exactly what they need to test. It also increases the chance that someone will inadvertently fail to implement certain necessary functionality. If you're outsourcing construction of the software, you can't expect the vendor's developers to accurately determine the unstated functionality from a use case description.

You will almost always have additional functional requirements that do not fit nicely into a particular use case. Earlier in this chapter, I mentioned the example of logging in to a system.

Clearly, that functionality must be implemented, but I don't consider it to be a use case. You might also have functional requirements that span multiple use cases. Consider the behavior the system should exhibit if a required connection to the Internet goes down. The Internet connection could fail while the user is executing any use case. Therefore, this error condition doesn't constitute an exception flow associated with a specific use case. It needs to be detected and handled in multiple operations. The analyst can place all the functional requirements that are not associated with or derived from a particular use case into the logically appropriate section of the SRS.

As an alternative to creating separate use case and SRS documents, you could select use cases as the organizing structure for the bulk of the functional requirements in the SRS. IEEE (Institute of Electrical and Electronics Engineers) Standard 830-1998, "IEEE Recommended Practice for Software Requirements Specifications," provides guidance on how to create an SRS (IEEE 1998). According to this standard, you might organize the functional requirements by system mode, user class, objects, feature, stimulus (or external event), response, or functional hierarchy. You could also combine or nest these organizing schemes. You might have functional requirements grouped by stimulus within user class, for example. There is no universally optimal way to organize your functional requirements. Remember that the paramount objective is clear communication to all stakeholders who need to understand the requirements.

I have found use cases to be a highly valuable technique for exploring requirements on many types of projects. But I also value the structured software requirements specification as an effective place to adequately document the functional requirements. Keep in mind that these documents are simply containers for different types of requirements information. You can just as readily store use cases, functional requirements, and other types of requirements information in the database of a requirements management tool. Just don't expect use cases to replace all your other strategies for discovering and documenting software requirements.