5

Use Cases

Use cases are a wonderful idea that has been vastly overcomplicated. Over and over again I have seen teams sitting and spinning in their attempts to write use cases. Typically they thrash on issues of form rather than substance. They argue and debate over preconditions, postconditions, actors, secondary actors, and a whole bevy of other things that just don't matter.

The real trick to doing use cases is to *keep them simple*. Don't worry about use case forms, just write them on *blank* paper, or on a *blank* page in a simple word processor, or on *blank* index cards. Don't worry about filling in all the details. Details aren't important until much latter. Don't worry about capturing *all* the use cases, that's an impossible task anyway.

The one thing to remember about use cases is: *tomorrow they are going to change*. No matter how dilligently you capture them, no matter how fastidiously you record the details, no matter how thoroughly you think them through, no matter how much effort you apply to exploring and analyzing the requirements, *tomorrow* they are going to change.

If something is going to change tomorrow, you don't really need to capture its details today. Indeed, you want to postpone the capture of the details until the very last possible moment.

Think of use cases as: *Just In Time Requirements.*

Writing Use Cases

Notice the title of this section. We *write* use cases, we don't draw them. Use cases are not diagrams. Use cases are textual descriptions of behavioral requirements; written from a certain point of view.

"Wait!", you say. "I know UML has use case diagrams, I've seen them."

Yes, UML does have use case diagrams, and we'll study them in a few pages. However, those diagrams tell you nothing at all about the content of the use cases. They are devoid of information regarding the behavioral requirements that use cases are meant to capture. Use case diagrams in UML capture something else entirely. And we'll discuss them in due course.

What is a use case.

A use case is a description of the behavior of a system. That description is written from the point of view of a user who has just told the system to do something particular. A use case captures the *visible* sequence of events that a system goes through in response to a *single* user stimulus.

A visible event is an event that the user can see. Use cases do not describe hidden behavior at all. They don't discuss the hidden mechanisms of the system. They only describe those things that a user can see.

The Primary Course

Typically, a use case is broken up into two sections. The first is the primary course. This section describes how the system responds to the stimulus of the user and assumes that nothing goes wrong.

For example, here is a typical use case for a point of sale system:

> *Check Out Item:*
>
> *1. Cashier swipes product over scanner, scanner reads UPC code.*
>
> *2. Price and description of item, as well as current subtotal appear on the display facing the customer. The price and description also appear on the cashier's screen.*
>
> *3. Price and description are printed on reciept.*
>
> *4. System emits an audible "acknowledgement" tone to tell the cashier that the UPC code was correctly read.*

That's the primary course of a use case! Nothing more complex is necessary. Indeed, even the tiny sequence above might be too much detail if the use case isn't going to be implemented for awhile. We wouldn’t want to record this kind of detail until the use cases was within a few weeks of being implemented.

How can you estimate a use case if you don’t record it’s detail? You talk to the stakeholders about the detail, without necessarily recording it. This will give you the information you need to give a rough estimate. Why not record the detail if we’re going to talk to the stakeholders about it? Because tomorrow the details are going to change. Won’t that chnage affect the estimate? Yes, but over many use cases those effects integrate out. Recording the detail too early just isn’t cost effective.

If we aren’t going to record the details of the use case just yet, then what *do* we record? How do we know that the use case even exists if we don’t write something down? Write the name of the use case. Keep a list of them in a spreadsheet, or a word processor document. Better yet, write the name of the use case on an index card and maintain a stack of use case cards. Fill in the details as they get closer to implementation.

Alternate Courses

Some of those details are concerning things that can go wrong. During the conversations with the stakeholders you’ll want to talk over failure scenarios. Later, it gets closer and closer to the time when the use case will be implemented, you’ll want to think through more and more of those alternative cases. They become addenda to the primary course of the use case. They can be written as follows:

> *UPC Code Not Read:*
>
> *If the scanner fails to capture the UPC code, the system should emit the “reswipe” tone telling the cashier to try again. If after three tries the scanner still does not capture the UPC code, the cashier should enter it manually.*
>
> *No UPC Code:*
>
> *If the item does not have a UPC code, the cashier should enter the price manually.*

These alternate courses are interesting because they hint at other use cases that the stakeholders might not have identified initially. In this case it is apparently necessary to be able to enter the UPC or price manually.

What else?

What about actors, secondary actors, preconditions, postconditions, etc. etc. What about all that stuff?

Don’t worry about it. For the vast majority of the systems you will work on, you won’t need to know about all those other things. Should the time come that you need to know more about use cases, then you can read Alistair Cockburn's definitive work on the topic: *Writing Effective Use Cases*, Addison Wesley, 2001. For now, learn to walk before you learn to run. Get used to writing simple use cases as above. As you master them (defined as having successfully used them in a project), you can ever so carefully and parsimonously adopt some of the more sophisticated techniques. But remember, don't sit and spin.

Use Cases Diagrams

Of all the diagrams in UML, use case diagrams are the most confusing, and the least useful. With the exception of the System Boundary Diagram, which I'll describe in a minute, I recommend that you avoid them entirely.

System Boundary Diagram

Figure 5-1 shows a System Boundary Diagram. The large rectangle is the system boundary. Everything inside the rectangle is part of the system under development. Outside the rectangle we see the *actors* that *act* upon the system. Actors are entities outside the system that provide the stimuli for the system. Typically they are human users. They might also be other systems, or even devices such as real-time clocks.

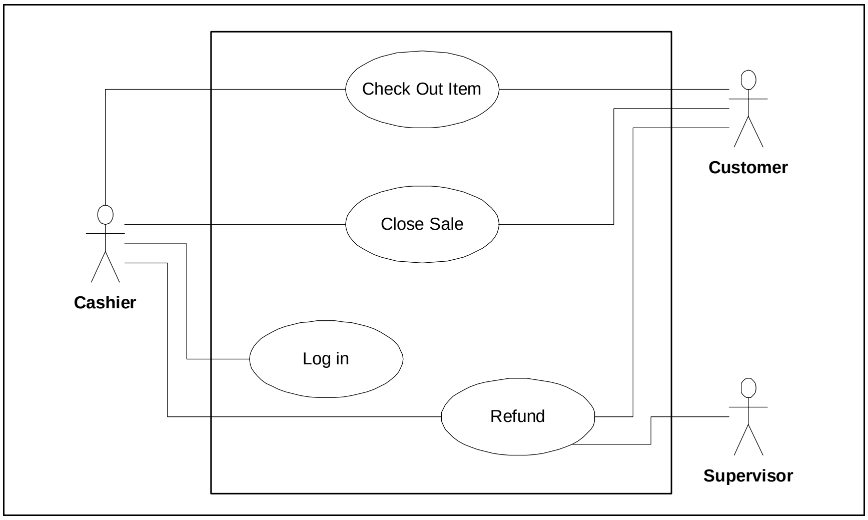

Figure 5-1
System Boundary Diagram

Inside the boundary rectangle we see the use cases. These are the ovals with names inside. The lines connect the actors to the use cases that they stimulate. Avoid using arrows, nobody really knows what the direction of the arrowheads means.

This diagram is almost, but not quite, useless. It contains very little information of use to the Java programmier, but it makes a good cover page for a presentation to stakeholders.

Use Case Relationships

Use case relationships fall into the category of things that “seemed like a good idea at the time”. I suggest that you actively ignore them. They’ll add no value to your use cases, or to your understanding of the system, and they will be the source of many never ending debates about whether or not to use «extends» or «generalization».

Conclusion

This was a short chapter. That’s fitting because the topic is simple. It is that simplicity that must be your attitude concerning use cases. If once you proceed down the dark path of use case complexity, forever will it dominate your destiny. Use the force, Luke, and keep your use cases simple.