SearchSoftwareQuality.com

From use case diagrams to context diagrams

Kevlin Henney

As long as practitioners recognize that use case diagrams are optional and iconic (as opposed to schematic), they shouldn't have problems. The diagrams are useful, for example, on whiteboards as a way of sketching and framing an agenda while people are writing up and reviewing use case detail on index cards.

The trouble starts, however, when projects end up with unreadable and overly complex use case diagrams. Those diagrams distract project members from the more useful endeavor of elaborating the use cases and result in wasted time. And the major value of the use case diagram -- showing the context of a software system -- ends up lost in a cloud of bubbles.

Use case distractions

There is little point in drawing a diagram if you don't know what you want from it. A common liability of use case diagrams is that the diagram diverts our attention from the real goal, which is to describe the use cases. As Alistair Cockburn noted in *Writing Effective Use Cases* (Addison-Wesley, 2001), under the heading "The Great Drawing Hoax:"

> For reasons that remain a mystery to me, many people focus on the stick figures and ellipses in use case writing since Jacobson's first book, *Object-Oriented Software Engineering* (1993) came out, and have neglected to notice that use cases are fundamentally a text form.

My previous article, "The pros and cons of use case diagrams," outlined the anatomy of use case diagrams, as well as highlighted both uses and pitfalls. In practice, many of the common pitfalls can be avoided by decomposing a monolithic use case diagram into a set of more focused diagrams or, alternatively, by not using use case diagrams at all and instead presenting one or more tables relating actors to goals. Although each of those options is more accessible than a single all-encompassing diagram, something important has been lost: the big picture.

A context diagram should offer a simple picture of how a system relates to the business or technology around it. It should not try to represent architectural partitioning or functional requirements.
,

To be precise, although a single use case diagram for the whole does indeed present a big picture, any simple message of how the system fits into the world is potentially lost in noise. On the other hand, breaking a single diagram into many or presenting the summary of actors and use cases in tabular form makes it difficult to see what actors employ and are employed by the system.

Context diagrams

A context diagram shows the software system and the world around the system. In this big picture view, the system is treated as a closed box. In essence, the goal of the development process is to both fill and fulfill the box. The emphasis is on how the system relates to the world, not on its architecture or any other form of decomposition.

Context diagrams have been around in one form or another for three decades and have been presented in various forms. Historically they were often shown as dataflow diagrams (DFDs), which follows naturally from their use in structured analysis methods that make heavy use DFDs as a formalism. In object-oriented modeling the appropriate style is to express the system and the world around it as objects connected by associations. Michael Jackson has taken the concept of context diagrams further and used them as the basis for understanding problem frames.

In UML, the simplest presentation of a context diagram shows the system as a box surrounded by its associated actors. The associations between actors and the system and between other actors are undirected, i.e. there are no arrows. The association is mutual and there is no notion of "flow." Notationally, such a diagram is a use case diagram without the use cases -- a "use-case-less" diagram. And although it's useful and legitimate UML, it is not a formal diagram type in UML.

Context diagrams are intentionally simple with a clear responsibility: They clarify where in the world a system fits, offering an opportunity to discuss the roles that use it and are used by it. It should be possible to fit a context diagram plus a brief description of each external role on a single page.

It is tempting to consider use case diagrams as context diagrams because they do indeed show context. However, use case diagrams show more than just a system and its context; they also show the behaviors the system is to fulfill. By contrast, a context diagram has the single, simple responsibility of showing the system and its context. It can complement more focused use case diagrams or tables of use cases. Keep in mind, of course, that none of these techniques is a substitute for use cases. They simply provide ways to frame, discuss and manage the detail.

More information on use cases and diagrams

How to document use cases

The pros and cons of use case diagrams

Use cases, scenarios and user goals

Advice for actors
Clarifying the role of the system inevitably -- and intentionally -- turns attention to the actors in the world around the system. It is worth keeping in mind that *actor* is a slight misnomer. In UML an actor is a role rather than something that plays a role. For roles carried out by humans the term *user role* is an increasingly common alternative term to actor. Although it's clearer, the term is not a drop-in substitute for *actor* because not all actors are human roles, and not all human roles would necessarily be classed as users in a context diagram. The common stick-figure notation for actors makes sense for human roles, but it looks a little strange for nonhuman roles such as devices and other software systems. For those kinds of actors consider using an alternative icon or simply a box stereotyped with «*actor*».

Leaving aside its personification as Father Time, time is a useful but often overlooked example of a nonhuman actor that can drive a system. For a wide-range of systems -- from real-time embedded systems to corporate accounting systems -- time can be considered an active agent that causes things to happen. Of course, what we mean by time and what we require of time is typically different for each of these kinds of systems:

- For a real-time system we may care about sub-millisecond accuracy and minimal drift for time-triggered events that are spaced milliseconds apart
- For accounting systems we care more about the daily and annual cycle of events, not about how the seconds pass

It is precisely those differences that are worth clarifying and why some explicit representation of time on a context diagram makes sense.

From a practical point of view, a time (or clock, calendar, etc.) actor avoids the problem of orphaned use cases. Use cases are expected to be driven by a primary actor, but if time is not an actor then time-triggered use cases are left floating.

Another practical aspect of representing time as an actor is that developers can see that time and related events can be mocked out for tests so that time is not treated as some global, given property of a software system, which is often assumed to be the case. Many systems have a time-driven aspect. If you are unsure whether to include a time actor, consider what would happen if you left the system alone for an extended period of time. If it does nothing, it is probably not time-driven.

Although it makes sense to describe existing hardware and software architecture as part of the context, keep in mind that context diagrams are not intended to be architectural diagrams. For example, one of the most common pitfalls is the inclusion of an actor named *Database*. If a new system is being developed, and the database is part of that development, then -- by definition -- the database is part of the system to be built and not external to it. In other words, the database in this case is not an actor and it lives inside the system.

On the other hand, if a database forms a part of the existing system context or is to be treated as a separate project, then it can be treated as external to the system of immediate interest. However, in this case *Database* is still not an actor. Actors are roles, so what is the role of the database? Does it provide customer information, system configuration information, pricing information? Name the actor after the role it plays, not the underlying technology. Where an actual database server fulfills more than one role it corresponds to multiple actors.

A context diagram should offer a simple picture of how a system relates to the business or technology around it, and it should not try to represent architectural partitioning or functional requirements. Its value lies in its simplicity, directness and sketch-ability.

About the author: Kevlin Henney is an independent consultant and trainer based in the UK. His work focuses on software architecture, patterns, development process and programming languages. He is a coauthor of *A Pattern Language for Distributed Computing* and *On Patterns and Pattern Languages*, two recent volumes in the Pattern-Oriented Software Architecture series. You may contact him at kevlin@curbralan.com.