Martin Fowler
martin_fowler@compuserve.com

Use and Abuse Cases

Use cases, despite the clunky name, have become one of the most popular techniques in object-oriented methods. Ivar Jacobson brought them to prominence by taking a widely used, yet informal technique and giving it a central place in system development. Since then nearly every writer on object-oriented design has talked about use cases and brought them into their approach. Naturally, they play an important role in the Unified Modeling Language (UML).

Use cases are valuable for several reasons. First, they help in discovering requirements. Use cases allow you to capture a user's need by focusing on a task that the user needs to do. Use cases help can help formulate system tests to ascertain that the use case is indeed built into the system. They also help control iterative development: in each development iteration, you build a subset of the required use cases.

Despite this wide usage, there are many problems about use cases that are not well understood. The basic definition—a sequence of actions that a system can perform, interacting with the users—is simple enough. But as soon as you start trying to capture the use cases for a system, you run into unanswered questions. I've also seen several projects get tangled up in use cases, getting into trouble in the process. So here are a few cases of abuses I've seen, and how I think you should avoid them.

Abuse by Decomposition

I've seen several projects get into difficulties by putting too much effort into structuring the use cases. Following Jacobson's work, the UML uses a pair of relationships between use cases: *uses* and *extends*. These can be useful, but I've more often seen them cause trouble—especially the *uses* relationship. In such cases, the analysts take a fairly coarse-grained use case and break it down into sub-use cases. Each sub-use case can be further broken down, usually until you reach some kind of elemental use case, which seems atomic to some degree.

This is functional decomposition, a style of design that is the antithesis of object-oriented development. It leads to problems in a couple of ways: The first is when this use case structure is reflected directly into the code, so that the design of the systems looks like the use cases. A common symptom is to find behaviorally rich controller objects manipulating dumb data objects, which are little more than an encapsulated data structure.

This kind of design loses most of the benefits of objects. The system duplicates behavior across the different controllers, and knowledge of the data structure is also spread around these controllers. The essence of the problem is that functional decomposition encourages you to think of a behavior in the context of a higher-level behavior. Done that way, it is difficult to use that same behavior in another context, even if it is mostly the same. With objects you want to think of behaviors that are usable in many contexts. Functional decomposition does not encourage that approach.

You can avoid this problem, of course, by remembering that the internal structure of the system does not need to look like the external structure (the use cases). However, this is a difficult concept for developers without object-oriented experience—who are also the most likely to build a functional decomposition.

Even if you don't find that the functional decomposition affects your internal design, a heavily structured set of use cases runs into trouble because people end up spending a lot of time on them. Driving every use case down to elemental steps, arguments about which use cases fit into which higher-level use case, all take time that could be better spent on other things. Capturing every detail of the use cases isn't needed in the early phases of development. You do want to look early at the areas of high risk, but a lot of the details can be left to the later stages of an iterative development. That's the whole point of iterative development.

So I discourage those new to objects from using the *uses* relationship. I also treat similarities between design and use cases as warning signs. The "extends" relationship seems to cause less trouble, but I still don't make a big thing of it. I've seen projects that used use cases effectively without either of these relationships, so they are by no means essential.

Abuse by Abstraction

Objects are all about abstraction. A good design is one that finds powerful yet simple abstractions, making a complex problem tractable. So as designers we are used to abstraction, encourage abstraction, glory in abstraction.

But abstraction can lead to trouble with use cases. I re-

Martin Fowler is an independent consultant based in Boston, Massachusetts.

member talking to a user at one project who confessed that although he understood the use cases at the beginning, he now felt lost with the more abstract use cases. "I think I understand them when the developers explain them to me, but I can't remember the explanations later." One of the primary purposes of use cases is to communicate with the users—the customers—of the system. Abstracting the use cases beyond the level of comprehension isn't going to help anyone. A lack of abstraction in the use cases does no harm, since the internal structure need not be the same as the external structure. I've not found that a lack of abstraction in the use cases leads to lack of abstraction in the internals. On the contrary, using less abstract use cases helps because mapping an abstract internal structure to a concrete use case helps us understand the abstraction.

Abstracting use cases can also lead to larger use cases, which, in an iterative development, are more difficult to plan with. You can also spend a lot of time arguing about the abstraction—time better used elsewhere.

So my advice is to err on the side of being too concrete. Above all, don't go more abstract that the user can follow. Use the concrete use cases to explain and verify your powerful abstractions.

Abuse by GUI

With all the GUI painting tools out these days, more people are using them to help determine the use cases. The logic is appealing. A GUI is concrete to a user; it helps snag the easy-to-forget details; it gives the user a sense of what the system will look like; GUIs are easy to prototype; they make reasonable demos to explain the capabilities of the future system.

I used to think that GUI prototypes were a good requirements tool for all these reasons. But there is a fundamental problem. When you show a GUI prototype to a user, it looks like nearly everything is done, that all that's left is a bit of wiring behind the scenes. Of course, we know that what lurks behind the scenes is the most complicated part of the exercise, but this is exceedingly difficult for customers to understand. GUIs lead to a false indication of progress and difficulty—it may be just a button on the UI, but providing it could take weeks of effort. There's a huge gap between the real effort and the perceived effort, making it difficult to do the negotiation that is so important in scope control.

And despite the fact that GUI tools seem so easy to use, there's always a lot of fine tuning to make it look just right. This fine tuning sets people's expectations in a particular direction, making people reluctant to make changes when a simpler design idea comes along.

So now I tell people to never show any GUI that hasn't got fully working code behind it. It's still good to mock up GUIs, but do it with pen and paper. That way you avoid the false indication of progress.

Abuse by Denying Choice

Take a high-end word processor. Think about the use-cases. Would you include *change a style, apply a style,* and *import styles from another document?* The problem with these use cases is that they don't directly address a user's needs. The user's real needs are something like *ensure consistent formatting within a document,* or *make one document look like another.* I've characterized the former as *systems* use cases and the latter as *user* use cases.

The problem with going directly to the system use case is that it denies you the chance to come up with other system use cases that would deal with this problem. You can't use only user use cases, however, because they don't fit well into the iterative scheduling process.

I'll confess that I haven't come up with a solid answer to this problem. I primarily put my attention to system use cases, for they are the most useful for iteration planning and system testing. However with every system use case I think about whether there is another user use case which is sitting behind it. I keep a note of these and try to come up with alternative system use cases.

Use cases are one of the most valuable techniques available to us. I use them all the time in my development work, and wished I had started using them earlier. But remember what you are using them for, and beware of these pitfalls. And somebody please write a good book on what makes a good use case—there's at least one active developer who really wants to read it.