# The  Misuse of Use Cases

(Managing Requirements)

In my experience I find that “use cases’ are more often misused than used correctly. The consequences of this misuse are often severe and strike at the core of good OO development. Several very common problems resulting from this misuse are:

- poor quality requirements
- poor quality designs that are nothing more than “functional decomposition” in “object clothing”
- wasted time and effort

The following story illustrates the last point above.

It was a very large project. Perhaps one of the largest OO projects ever undertaken. At one point I heard a senior manager comment that there were over one thousand software developers, spread over three continents, working on the project. The project involved the near-simultaneous development of a multi-level framework with numerous applications built on that framework - pretty aggressive, heady stuff for a team new to OT!

For the size and complexity of the project, there were surprisingly few false starts and setbacks.  One anecdote does, however, stand out in my mind.  A Sherlock Holmes fan on the project refers to it as:  ***The Case of the Useless Use Cases***. Early in the project, a team was assembled at corporate headquarters to begin developing  the framework.  The initial team activities focused on

- building domain models and
- debating domain specific architectural issues known from experience with similar legacy systems to be issues that would have to be addressed within the new OO framework.

As work on the framework progressed, the architecture team realized they needed to understand the scope of the requirements for applications that would be built using the framework.  So, since “use case” was a fashionable buzz word at the time, a call was sent out to appropriate application development teams to “ send us your use cases". In response, application teams at corporate sites around the world purchased Ivar Jacobson's book**: Object-Oriented Software Engineering, A Use Case Driven Approach**; studied up on the vending machine example and wrote use cases - thousands of them! The final tally made by the framework team at corporate headquarters was something like 12,386 use cases.

Unfortunately these use cases were not very useful to the framework team - not because they didn't need the information in them - but because the information was at the wrong level of abstraction. A quick back-of-the-envelope calculation estimates the bottom line cost of producing those 12,386 useless "use cases" to be over a half of a million dollars. Staggering isn't it? Furthermore, the morale cost to the application teams was high and their confidence in the framework team was eroded. Team members had invested a lot of time and effort in learning about, and developing, use cases only to have their work ignored. The really sad part is that in circumstances like this, the original use cases are typically not only too detailed, they are usually full of errors. Later in the project when these detailed use cases are needed, they almost always have to be redone. This incident is just one of many I have witnessed that occurred because the requirements process was not well understood nor properly managed.

Consider Figure 1 for a moment. The left hand column deals with business requirements and interface specifications. The right hand column deals with development deliverables such as software designs and source code. It is well understood and accepted that a number of levels of abstraction are required in the right hand column. Few development teams would start out a multi-year project by jumping right into writing the source code. Even the most undisciplined teams will do some level of domain analysis and design before they start coding.

What is not so well understood or accepted is that a number of levels of abstraction are required in the left hand column. It is no more possible to get correct requirements by starting at the detailed specification level than it is to write well designed, correct systems by starting at the level of the source code! There are fundamental principles of requirements gathering that cannot be ignored without serious consequences to a project.

| **Requirements Artifacts** | **Development Artifacts** |
|---|---|
| Business requirements<br>first few levels of use cases | domain models |

| | |
|---|---|
| interface specifications<br>level at which interface bindings are introduced | application models |
| | architectures |
| more details<br>several more levels of use cases | detail designs |
| complete detailed specifications<br>final level of use cases | source code |

Figure 1

I can’t cover all the relevant topics in the space allowed, but I would like to make at least the following points in the rest of this article.

- Requirements should be organized hierarchically.
- Hierarchical classification of use cases, need not, and should not become functional decomposition.
- Business requirements should be kept separate from interface specifications
- Do not directly derive your design from your use cases.
- There is a minimal set of required fields for a well written use case.

**Requirements should be organized hierarchically.** There are several reasons for this. First and foremost is that a hierarchical organization is needed to manage the complexity of the typical set of requirements. Humans have a notoriously limited ability to deal with complexity. If team members, clients, and users are to be able to understand the requirements, reason about them, debug them, and use them to validate development products, the requirements must be hierarchically structured. Our rule of thumb is that the top level requirements for the system should be expressed in no more than a dozen or so use cases. No layer of use cases should have no more than five to ten times the number of uses cases than in the next higher layer.

A second reason for having uses cases at several different levels of detail is so that there is a complete set of requirements at the appropriate level of detail for "testing" each of the domain, application, architectural, and detail design models. This implies that each level of "use cases" must be complete in the sense that all categories of system requirements are covered at each level. Level N+1 should not introduce any new category of requirements, but should simply divide existing categories into subcategories. Thus level N+1 will have a larger, more detailed, set of use cases than level N.

Use cases are best developed iteratively and incrementally, the same way as the rest of the system deliverables.

**Hierarchical classification is not functional decomposition.** Use case 1.1 is NOT the first step of use case 1. Use case 1.1 is a specific, more detailed, complete use case within the category of use cases defined by use case 1.

**Keep business requirements separate from interface specifications.** In his book on use cases, Jacobson uses the example of a vending machine. One use case described is that of purchasing a item from the vending machine. The example use case details the specific interface mechanisms such as: insert coin into slot. There is no problem per se with this example as long as the fundamental business requirements have previously been expressed in interface neutral terms (e.g. accept payment from customer). The problem arises when the first level of requirements jump directly to interface specifications. Unfortunately, this is precisely the way most OO projects teams think they should be doing things. They neglect fundamental principles of requirements gathering in the name of "use cases." This does not serve the project well. Unless the first level of requirements are interface neutral, clients are robbed of the natural opportunity to consider other alternatives and designers are not prompted to build extensibility into the system. Conversely, when the first several levels of requirements are interface neutral and the point at which the interface binding is introduced is explicit, users have a deeper understanding of the true nature of their needs and are better able to articulate true requirements. For example when it is explicit that "deposit coin" is simply an interface binding to the requirement of "accept payment," software designers, hardware engineers, marketing personnel, product managers, et. al. are prompted to consider the possibilities and implications of other interface bindings such as an electronic cash or credit card reader.

**Do not directly derive your design from your use cases.** If you do, "use case development" simply becomes an excuse for functional decomposition. Use cases stop at the system interface boundary! Use cases should describe sequences that actors follow in using the system, but use cases MUST NEVER specify what steps the system takes internally in responding to the stimulus from an actor.

A software system is a specific instantiation of an architecture customized to satisfy a specific set of functional requirements. The architecture was not chosen or created so much because of the functional requirements, but because of standard domain relationships and other system requirements such as time, space, reliability, distributability, portability, extendibility, standardization, etc. We are just emerging from the dark ages of ad hoc design, into the future of architectures created from customizable yet standard frameworks and patterns. Let's not take a giant step backward into the age of one-of-a-kind designs based on functional decomposition. If I catch another team doing this in the name of "a use case driven approach," I'm likely to bring the whole team blindfolded to face a firing squad for a public execution at the next Object Expo conference!

Standard architectures are being crafted to meet classes of non-functional requirements. The way in which an architecture is instantiated to implement a set of functional requirements is documented by a set of object interaction diagrams, NOT by a set of use cases.

**There is set of required fields for a well written use case.** A free template is available from our web page at www.qualsys-solutions.com.

Organizing requirements into discrete use cases is a good idea. I'm an enthusiastic supporter of use cases, but I'm dismayed at how often teams misuse use cases. The requirements gathering process is one of the most difficult yet important parts of software development. Manage it carefully.