

# The Misuse of Use Cases

## (Managing Requirements)

Dr. Timothy Korson

In my experience I find that "use cases" are more often misused than used correctly. Worse still, 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 realised 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 buzzword 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 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 neither well understood nor properly managed.

Consider Figure 1 for a moment. The column on the left 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 nor accepted is that a number of levels of abstraction are also required in the column on the left. 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<br>bindings are introduced | application models    |
|   | architectures         |
| more details<br>several more levels of use<br>cases                             | detail designs        |
| complete detailed<br>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 organised hierarchically.
- Hierarchical classification is not functional decomposition.
- Keep business requirements separate from interface specifications.
- Do not directly derive your design from your use cases.
- There is a set of required fields for a well-written use case.

**Requirements should be organised hierarchically.** There are several reasons for this. First and foremost is that a hierarchical organisation 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 more than five to ten times the number of uses cases 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 could divide existing categories into subcategories. If a lower level, a detailed use case is found that does not fit into an existing category, all higher levels of the use case hierarchy should be updated to reflect the new category.

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, 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 an 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 jumps 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 is 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," then 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 customised 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, standardisation, etc. We are just emerging from the dark ages of ad hoc design, into the future of architectures created from customisable 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 public execution by a firing squad 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 and white paper is available from our web page at [www.korson-mcgregor.com](http://www.korson-mcgregor.com).

Organising 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.

# Configuring A Use Case Process

## (Managing Requirements, Part 2)

I am a strong proponent of use cases, yet as I pointed out in my last column, I find that use cases are often misused. In this column I would like to address the configuration of the use case process, which I find to be equally misunderstood and misused.

An international corporation I am working with is developing a distributed, multi-currency, and multi-lingual, core financial system. In many ways the project exemplifies the best current practices for OO software development.

During the first six weeks we developed a comprehensive domain model. This required an extensive interview process with the company's CPAs, auditors, treasurers, controllers, and other financial personnel. The domain model is stored in a commercial CASE tool as an integrated set of seventeen UML class diagrams. To validate the domain model, a video was made of a walkthrough of the diagrams. The video was then distributed internationally to several hundred of the company's key financial personnel. The blown up versions (required for the video) of the domain model class diagrams were affixed to the walls in the project development area, and are frequently referenced and occasionally updated by the development team.

The overall software architecture (see **figure 1**) is built on fundamental OO principals and reuses a standard database architecture that is described in detail in Peter Heinckiens new book. The database architecture maps from relational tables to objects in such a way that the business model is entirely separated from the database technology. This allows easy migration to other database technologies. A professional integrated Java development environment is being used to create the actual software, and commercial class libraries play an important role in reducing development effort. Experts in the field are consulted for important design decisions, and a technical issues resolution database is maintained. It details the pros and cons of the alternatives debated as well as the rationale for the design selected. Seventeen system increments are defined in the project plan. As each increment is finished, it is tested on multiple platforms and with multiple commercial database management systems. A corporate steering committee was set up to oversee and facilitate the project. Beta test sites have been selected, and there is an excellent working relationship between the development team members and the financial personnel that will ultimately use the software system. When ambiguities in domain knowledge or application requirements arise, a financial expert is available to dialogue with the team within an hour or two. This is ideal. The financial personnel have a vested interest in the project and usually see any problem through to its resolution.

We are well into the development of this system, yet so far we have only defined eighteen use cases, and none of these have been elaborated into complete interaction dialogs or put into a standard UML detail use case template.

This lack of a complete set of detailed use cases is heretical to many of my colleagues. I agree that in most cases much more effort should be devoted to use case development, but not all projects require the same use case process.

### A Use Case Process Should be Configured for Each Specific Project

A full-blown use case process requires a complete set of hierarchically organised use cases, developed in sufficient detail so that the system test cases can be directly derived from the most detailed level of use cases. These use cases should be completely elaborated with alternative paths, preconditions, etc. A use case model should be developed that show the extends and uses relationships. For each use case there should be a sequence diagram that traces the use case to the implementation.

Large, multi-site, multi-team, projects often need a full-blown use case process, but not all projects fall into that category. There are several factors that affect the configuration of a use case process.

**Multi-team.** The larger the number of teams on a project, the greater the need for detail and formality in the use case process. Often use cases cross team boundaries. Well-written, detailed use cases can

document certain aspects of team interactions and help avoid misunderstandings. At some of our consulting sites we have augmented the use case template and corresponding sequence diagram to explicitly show team boundaries. *The financial system described in the opening paragraphs of this column is being developed by a single team, with all team members located in a dedicated project work area.*

**Multi-site.** Multi-site projects compound the problems of multi-team development and make it even more necessary for all project documentation--including use cases--to be formal and detailed.

**Specialised Teams.** I favour cradle-to-grave integrated teams, but work with many companies that have separate teams for requirements analysis, design and development, and testing. Obviously when the requirements team is separate from the development team, the need for documentation detail is increased. *The financial system team is tightly integrated. The same individuals who are writing the Java code interviewed the accountants, created the domain model, and will test the system.*

**Team Member Domain Knowledge.** When I was a professor at Clemson University, I spent my summers as a visiting scientist at the Software Engineering Institute. I will always remember one particular speaker at the SEI monthly seminar series. He worked with radar signal processing, and the thesis of his presentation was that it is easier to teach a radar specialist how to program than it is to teach a programmer about radar technology. He gave several relevant and amusing anecdotes to illustrate his point. Most of his observations are difficult to refute--which is why I am so adamant that development teams participate in the domain analysis. The less a team knows about the domain, the more details must be spelled out in the use case model. *Many of the team members in the financial system example have degrees in accounting and have extensive financial backgrounds.*

**Integration with the System Test Process.** If the use case model is tightly integrated with the system test process, the cost of developing detailed use cases is repaid when the system test cases are developed. If a separate independent test team (which would have to develop test cases anyway) is involved in developing the use cases early in the project lifecycle, then many benefits accrue. In this case, the independent test team can be involved in domain model validation and other QA activities throughout the project lifecycle. *For the financial system there is no independent test team.*

**Extent to Which the Requirements Already Exist in Some Other Form.** If you are starting a large multi-site project for which there are few existing written requirements, I would recommend that the requirements gathering process include the development of a complete detailed set of use cases. If the project is a reengineering of an existing system, or if for some other reason the project already has an existing set of requirements, the team must decide what percentage of the requirements to rewrite as a hierarchically structured set of use cases. Many factors enter into this decision. I recommend always creating at least the top two to three levels of domain use cases to help drive validation of the domain model. When requirements already exist in some form, most teams choose to develop only the next few levels of use cases, resulting in 50 to 200 use cases. This is reasonable. In the presence of an existing detailed requirements document, the incremental value of each additional level of use cases decreases dramatically. An exception to this is when the use case process is tightly integrated with the system test processes, and the independent test team derives their test cases directly from the detailed use cases. *Business requirements for the financial system already exist in several large manuals, as well as in several legacy systems.*

**Stability of the Requirements and Availability of the Clients.** Uses cases must be kept accurate and updated as requirements change. If the requirements are fairly stable, the task of maintenance is manageable. However, if the requirements are subject to rapid change, the cost of updating thousands of use cases may be prohibitively high. In some instances, when the clients are readily available for input and rapid feedback, the team may elect to forgo written detail interface specifications in favour of a continuous client review process of actual project increments. *Our interface specifications are not well defined, but our clients are readily available for input.*

Many other factors, including the nature of the application, level of automation of the testing process, availability of resources, time to market, corporate culture, etc., should influence the configuration of a use case process. The fundamental point I wish to make is that one size does not fit all, yet most project managers seem to think that if they are going to use objects, there is one, and only one, right way to utilise use cases.

My experience leads me to believe in a use case spectrum (see **figure 2**). Your project is likely somewhere in the middle of the spectrum. Configure your use case process intelligently and then manage it carefully.

| <b>Heavy-Weight Use Case Process</b>          | < |  | > | <b>Light-Weight Use Case Process</b>              |
|---|---|--|---|---|
| Multiple Sites<br>Multiple specialised teams  |   |  |   | Single Site<br>Single integrated development team |
| Development team has limited domain knowledge |   |  |   | Development team has extensive domain knowledge   |
| Integrated testing process                    |   |  |   | No external testing group                         |
| Few pre-existing written requirements         |   |  |   | Extensive pre-existing written requirements       |
| Stable specification                          |   |  |   | Rapidly changing interface specifications         |
| Limited access to clients                     |   |  |   | Ready access to knowledgeable clients             |

**Figure 2**

## Constructing Useful Use Cases (Managing requirements - part 3)

"Software Engineers are useless. I'd rather hire radar specialists and teach them how to program than hire programmers and try to teach them about radar signal processing." This rather strong statement was spoken with conviction by the manager of large government project after a software system in his department incorrectly warned of an incoming ICBM missile! The manager's frustration was exacerbated by the fact that the programmer would not accept any responsibility for the incorrect functioning of the system. The programmer claimed that it was not his fault, but the fault of incomplete specifications. The manager acknowledged that the requirements document did not treat the specific circumstances that led to the incorrect warning, but felt that any programmer working in his department should have had enough basic knowledge about interpreting radar data to read between the lines of the requirements document. "No radar specialist would have made such a basic error" stated the project manager.

It is my experience that the major problems in systems development are not technical. The most serious problems are to do with requirements - getting the right requirements and getting the requirement right.

The larger and more complex the system being developed, the more that requirements problems become among the highest risk factors. This is true regardless of the domain. I have personally observed this within aerospace, telephony, financial systems, inventory control, and a wide variety of other domains.

A major point of this column is that simply using use cases to specify requirements does not automatically mean that you will get good requirements. Use cases are no more than a structured format for gathering and expressing requirements. A good format is helpful but not sufficient. Identifying a complete set of actor roles means that I will capture all of the user's viewpoints, but what if some of the actors don't really understand the true business needs? What if the development team misunderstands the use cases? In both cases the developed system will not meet the needs of the organisation.

It is my hypothesis that good use case development relies on the knowledge gained during domain analysis. Conversely, to understand a domain, it is helpful to know the actors and the major domain activities. This implies that an iterative process that includes both domain analysis and use case development is essential to getting good requirements. I will explore the fundamentals of performing useful domain analysis in a future article. Briefly, domain analysis is a development phase where the fundamental domain concepts, the essential attributes and behaviours of each concept, and the static structural relationships among concepts is documented. UML is the standard notation for capturing a domain model. Note that use case development is function oriented; domain analysis is concept oriented. A domain model does not have the notion of a software system. The "classes" in the domain model are pure domain concepts.

I am astounded at how many projects skip domain analysis. I've seen all too many projects with a detailed set of use cases, and a large number of design level class diagrams, but no true domain models. In such cases I am always willing to bet that not only is the design sub-optimal, but that the uses cases are untrustworthy and probably not well understood by the development team.

**You can't create correct, useful use cases if you don't understand the domain.** This is as true for the client as for the development team. Never assume the client knows and can articulate real business needs. How many times have you correctly implemented a set of specifications only to find that the client doesn't like the delivered system because the specifications they wrote didn't really reflect their needs? During the process of domain analysis, clients are forced to clarify their understanding of the domain. Domain analysis facilitates understanding in several ways. Typically each actor and domain expert has a very limited view of the domain in which an application will live. For example, in a hospital, nurses have one point of view, doctors another, lab technicians another, and administrators yet a different view. Having a representative from each group participate in concept definition and modelling enriches the understanding of everyone. We find that, invariably, as actors, clients, and domain experts participate in creating the UML domain level class diagrams, the team will find they need to go back and change previously defined use cases. As the domain models near completion, we frequently hear clients comment that: "I never before really understood how this all fits together." In some cases we find that requirements documents get substantially rewritten during and after conducting domain analysis.

**You can't implement correct use cases correctly if you don't understand the domain.** This goes back to the anecdote above. Another illustration comes from a current accounting project. The use cases call for an accountant to be able to print a journal listing. It turns out that the term *journal* has a number of different meanings to accountants. It could be the physical book where raw transactions were historically recorded. It could be simply a group of transactions that are posted together. It could simply refer to the type (e.g. cash, sales) of transaction being recorded. Without a fairly complete understanding of accounting concepts, a software engineer will misinterpret even well written use cases.

**You can't get the uses cases totally correct at the beginning of the project, because your understanding of the domain matures as you move through the project life cycle.** In the same way that one's understanding of true requirements grows as one participates in domain analysis, one's understanding of the domain grows as the system is designed and implemented. This means that to get useful use cases, all phases of the project must be involved in the iterative/incremental lifecycle. I've seen a number of project managers who thought they were doing good iterative/incremental development when, in fact, they were only iterating over detail design and implementation. A good project management document will plan to revisit use case development, domain analysis, and architectural design as well as detail design and implementation.

**Don't try to write very many use cases before doing the first cut at domain modelling.** During domain modelling the basic domain vocabulary is clarified. It is not useful to delve very deeply into functional requirements until fundamental concepts are understood. I recommend identifying all the actor roles and a handful of essential use cases for each actor before starting domain analysis. Don't go any deeper than the top level of uses cases. If other project constraints (e.g. contractual requirements) dictate getting a complete, detailed, set of requirements up-front, then simply be prepared to modify the requirements as the project lifecycle progresses.

**The top-level use case template should contain a "why" section.** Written specifications are always incomplete, in part because human language is imprecise, and in part because the use case author makes certain assumptions about what the reader already knows about the domain and the context of the given use case. I will explore and illustrate this point in a future article.

**Good software engineering is NOT use case driven!** Requirements are important. Use cases are a good way to structure requirements, but if you care at all about component based development, reuse, robust distributed architectures, cost, schedule, etc., than you cannot afford to let any single viewpoint drive your project. I count Ivar Jacobson as a friend. I respect his technical expertise and I have found good advice in his writing. I believe, however, that many practitioners have misunderstood and misapplied his advice because they have focused on catch phrases. I have been tempted in the past to say that good software engineering is driven by the concepts in the domain class model, but that is as wrong as saying that good software engineering is use case driven. Good software engineering is driven by a number of concerns that are weighted differently by different organisations and different projects within an organisation. These concerns include: technical design considerations, user requirements, reuse, modifiability, performance, standardisation concerns, schedule pragmatics, and other business drivers. Each project should be driven by a custom weighted vector of considerations. In each case, however, I think it is accurate to say that a project should be as much driven by domain concept modelling as it is driven by use case development.

**Developers who don't care about understanding the domain are pretty useless.** This follows from the above discussion. I just wanted to say it straight out and in bold print. I'm not trying to say that all developers need to become domain experts or have domain analysis skills equal to their development skills. Technology experts are very valuable. What I am saying is that the days of exclusive compartmentalisation are gone. Cradle-to-grave teams are the most productive software development organisations. Within a team, I would expect individuals to specialise, but I would also expect each member to at least participate to some degree in most of the phases. Specifically, I would expect all team members to understand and be able to describe the project domain model in detail. This would not solve every problem of a developer misunderstanding the intent of a set of use cases, but it would clear up a great many of them.