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 have 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 organization.

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 behaviors 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 modeling 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 meaning 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 on 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 modeling. During domain modeling 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[1] 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 organizations and different projects within an organization. These concerns include: technical design considerations, user requirements, reuse, modifiability, performance, standardization 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 modeling 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 compartmentalization are gone. Cradle-to-grave teams are the most productive software

development organizations. Within a team, I would expect individuals to specialize, 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.

---

[i] The Misuse of Use Cases, Object Magazine, May 1988, pp. 18-20