Use Case Pitfalls:
Top 10 Problems from Real Projects Using Use Cases

Susan Lilly
SRA International, Inc.
4300 Fair Lakes Ct.
Fairfax, VA 22033
susan_lilly@sra.com
703-227-5103

Abstract

One of the beauties of use cases is their accessible, informal format. Use cases are easy to write, and the graphical notation is trivial. Because of their simplicity, use cases are not intimidating, even for teams that have little experience with formal requirements specification and management. However, the simplicity can be deceptive; writing good use cases takes some skill and practice. Many groups writing use cases for the first time run into similar kinds of problems. This paper presents the author's "Top Ten" list of use case pitfalls and problems, based on observations from a number of real projects. The paper outlines the symptoms of the problems, and recommends pragmatic cures for each. Examples are provided to illustrate the problems and their solutions.

Introduction

Over the past few years, we have seen a number of projects make their first attempts at developing use cases. These projects have used use cases in a number of ways: as the entire system requirements specification, as part of the system requirements, as an analysis technique to elicit user requirements that were subsequently specified in other forms (e.g., traditional "shalls"), and as software subsystem-level requirements. The project teams that developed the use cases have included developers and/or analysts; in some cases the project teams have included customers or end users as well.

Although the project teams had little trouble getting started with use cases, many of them encountered similar problems in applying them on a larger scale. These problems include undefined or inconsistent system boundary, use case model complexity, use case specification length and granularity, and use cases that are hard to understand or never complete. These have been grouped and summarized here as a *"Top Ten"* list of use case pitfalls and problems, which may be encountered by inexperienced practitioners.

A sample problem is used to provide simple examples throughout this paper. *The Baseball Ticket Order System* is a computer system that is to be deployed to simplify customer sales for baseball games. Customers may view the season schedule and reserve tickets at kiosks placed in convenient locations, such as malls. Alternately, customers may call an 800 number and a phone clerk will reserve tickets for them. The customer may pay by credit card, or may pay at the time the tickets are picked up at the stadium on the day of the game.

The Top Ten List

Problem #1: The system boundary is undefined or inconsistent.

Symptom: The use cases are described at inconsistent system scope -- some use cases at business scope, others at system or even subsystem scope.

One element of the use case model is a labeled box that indicates the system boundary; the actors go outside of this box, and the use cases go inside. Before we determine the actors and use cases, we must be explicit about what we mean by "system." Is it a computer system? An application? A component or subsystem? Or is it a whole business enterprise? Use cases might be used to describe any of these "system" boundaries, but should only focus on one at a time. The actors and use cases appropriate at one system boundary are likely to be incorrect for a different system boundary. A common problem is the mix of both scopes in the same use case model, or even within a single use case specification.

Example: A *Kiosk Customer* uses the computer system to order tickets. Alternately, a *Phone Customer* may call the ticket business, and a *Phone Clerk* (an employee of the ticket business) may use the computer system to order tickets. Who are the actors? Figure 1 illustrates a mixed-up system boundary: The modelers have tried to show both the users of the business and the users of the system in the same use case model.

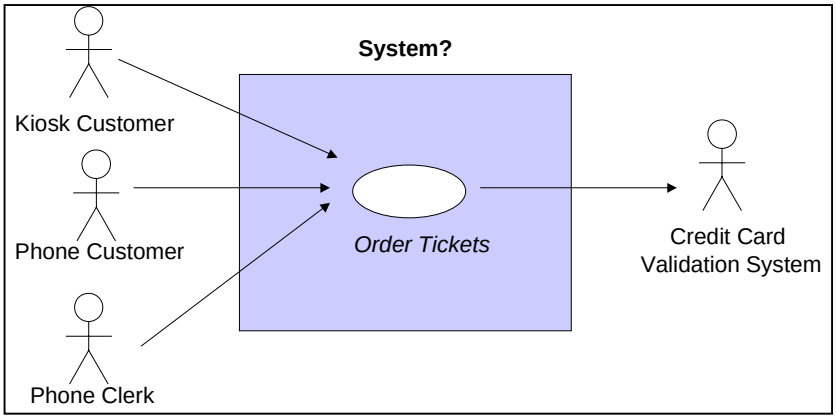

Figure 1: Use Case with Mixed-Up Scope

Cure: Be explicit about the scope, and label the system boundary accordingly. Say: "Yes, the business model is very interesting, but right now we are defining our use cases at the computer system scope" -- and then stick to it. *Example:* In Figure 2, the system boundary represents a computer system, and *Kiosk Customer* and *Phone Clerk* are actors who use the **Order Tickets** use case.

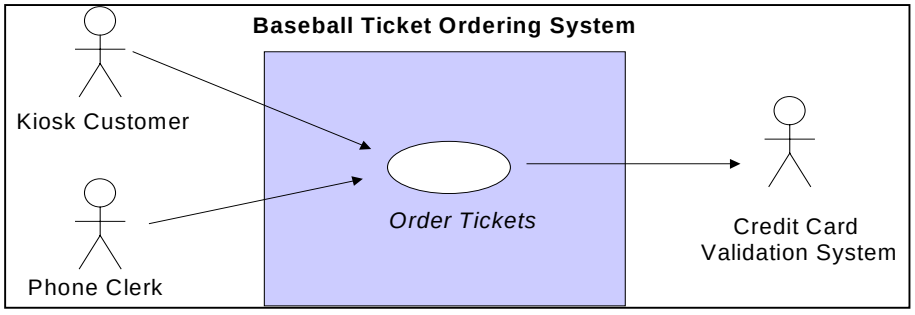

Figure 2: Use Case at Computer System Scope

In Figure 3, the system boundary represents a whole business enterprise. The actor, *Phone Customer*, is a user of the ticket business, but is not a user of the computer system. Both of these are appropriate ways to model; the choice between them depends on whether we are trying to define the requirements of a computer system (use Figure 2), or using use cases in business process modeling or reengineering (use Figure 3).

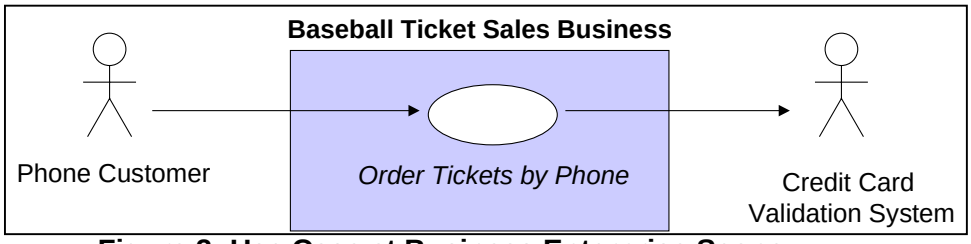

Figure 2: Use Case at Business Enterprise Scope

Symptom: Looking at the use case model, it's not really clear what's inside and what's outside the system. This problem often comes up when the use cases are modeled using a visual modeling/CASE tool (including the leading one on the market) that doesn't show the system boundary on the use case model.

Cure: Draw the system boundary (at least in your head). If the modeling tool does not draw a system boundary, place the use cases inside and the actors outside an imaginary box. *Example:* Figure 4 shows the same use case model, formatted in different ways. The model on the left has mixed up the actors and use cases; the one on the right has placed the use cases in the middle ("inside") with the actors on the "outside."

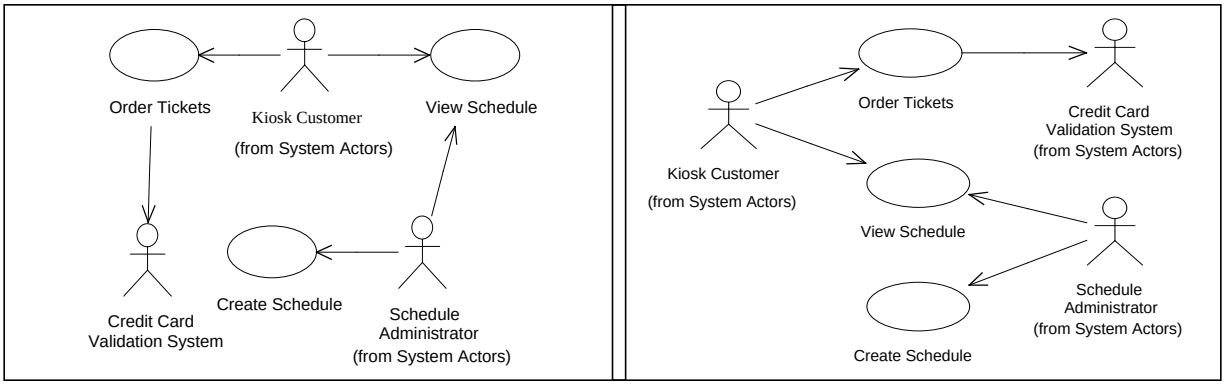

Figure 3: Use Case Model Formatting (bad and better)

Problem #2: The use cases are written from the system's (not the actors') point of view.

Symptom: The use case names describe what the system does, rather than the goal the actor wants to accomplish.

Cure: Name the use cases from the perspective of the Actor's goals. *Example:* **Process Ticket Order** and **Display Schedule** are things the system does (bad use case names). **Order Tickets** and **View Schedule** are goals of the system's users (good use case names).

Symptom: The steps in the use case specification describe internal functionality, rather than interactions across the system boundary.

Cure: Focus on what the system needs to do to satisfy the actor's goal, not how it will accomplish it.

Symptom: The use case model looks like a data/process flow diagram.

Cure: Watch out when the use case model includes use cases that are not directly associated with an actor, but are associated with <<***uses***>> or <<***extends***>> relationships. Sometimes this is an appropriate way to model the use cases. But many neophyte use case modelers (especially those who are programmers, or who have a process modeling background) misuse these associations, functionally decomposing the problem, rather than focusing on the interactions between actors and the system. Take a look at the specifications of used or extension use cases, to ensure that the steps in them describe interactions between the actor (of the base use case) and the system. If the steps are entirely focused on internal processing, the used and extended use cases are probably being used as a mechanism for functional decomposition. (If so, they don't belong in the use case model.)

Problem #3: The actor names are inconsistent.

Symptom: Different actor names are used to describe the same role. This is amazingly easy to do, since different sources of requirements often use variant names for the same thing -- and similar names for quite different things. When a problem is large, there are often multiple teams working on use case models for different parts of the problem, and the same (logical) actor may appear with variant names from model to model. *Example:* The role of the person who manages the online baseball schedule is called "Schedule Administrator" in one model, "Schedule Manager" in another, and "Scheduler" in a third.

Cure: Get agreement early in the project about the use of actor names (and other terms). Establish a *glossary* early in the project and use it to define the actors. The glossary should specify the actor name, its meaning, and any aliases that this name is known by. Include the glossary as an appendix to the use case document.

Problem #4: Too many use cases.

Symptom: The use case model has a very large number of use cases.

Cure: Make sure that the granularity of the use cases is appropriate. Use cases should reflect "results of value" to the system's users -- the attainment of real user *goals*.

- Combine use cases that describe trivial or incidental behavior that are actually fragments of the real use cases. Use cases are sometimes chopped into fragments when there is an attempt to associate user interface screens to use cases in a 1-to-1 relationship.
- Remove use cases that describe purely "internal" system processing ("internal" with respect to whatever system boundary is being used).

Example: In Figure 5, the Happy *Kiosk Customer* actor is associated with a use case called **Order Tickets** -- the customer's real goal in walking up to the kiosk in the mall. The Sad *Kiosk Customer* actor is associated with three different use cases. They all describe interactions between the *Kiosk Customer* and the system, but they represent incidental steps in the attainment of the actor's real goal (to order tickets). How did the "real" use case get split into three sub-goal use cases? The modelers were attempting to make a separate use case for each user interface element.

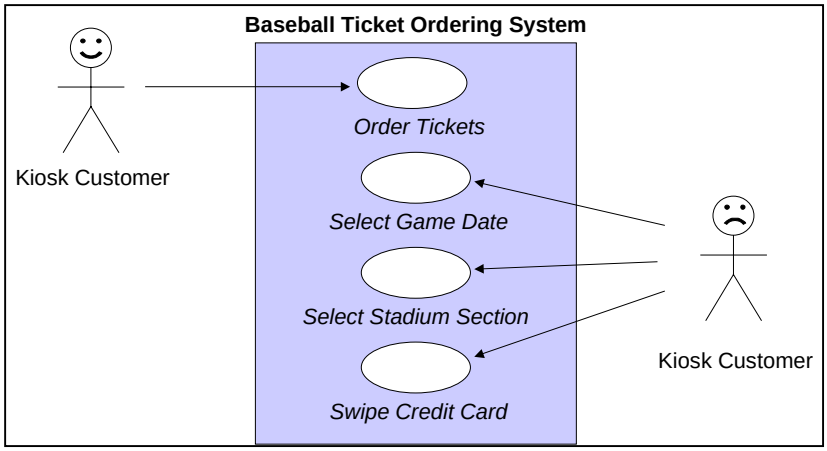

Figure 4: Real Use Cases vs. Incidental Actions

If the granularity of the use cases is right, but the system is simply very large, partition the set of use cases. Break the use case model into use case *packages*, each of which contains a cohesive set of use cases and a limited set of actors. *Example:* Figure 6 shows a use case model that has a large number of use cases. Figure 7 illustrates the same set of use cases, partitioned into 5 packages. Each package should contain a "cohesive" subset of the use cases, grouped around one or more actors who share common goals.

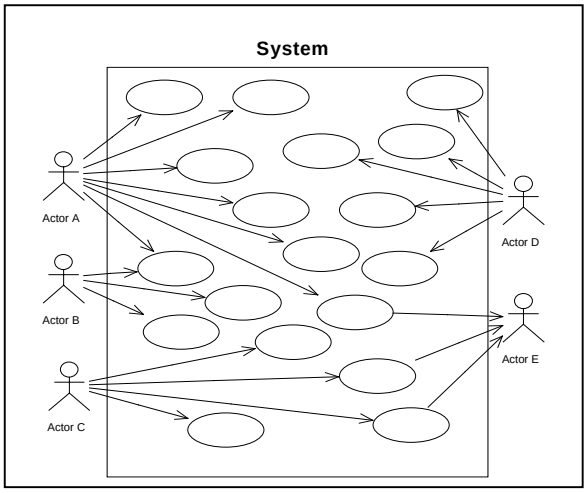

Figure 5: Model Needs Partitioning

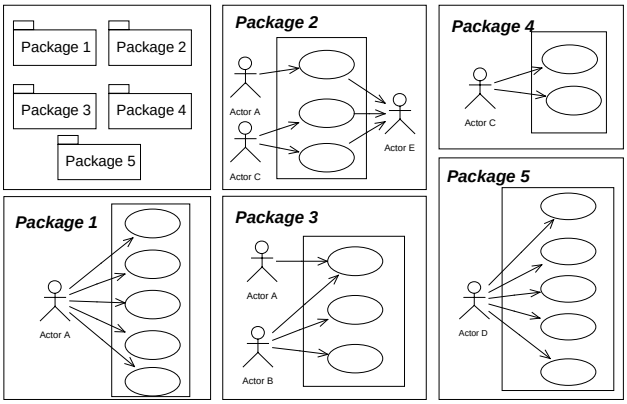

Figure 6: Model with Packages

Problem # 5: The actor-to-use case relationships resemble a spider's web.

Symptoms: (a) There are too many relationships between actors and use cases. (b) An actor interacts with every use case. (c) A use case interacts with every actor.

Cure: The actors may be defined too broadly. Examine actors to determine whether there are more explicit actor roles, each of which would participate in a more limited set of use cases. *Example: Employee* is very general, and is associated with a large number of use cases.

Phone Clerk and *Schedule Administrator* are more specific; each of these is associated with a smaller, more role-oriented set of use cases.

There may be cases where recognition of a more general class of actors helps to simplify a model. This often occurs where two or more actors are associated with the same set of use cases, because of some commonality in their roles. The resulting use case model has a spider's web of crossed lines between actors and use cases, as shown in Figure 8.

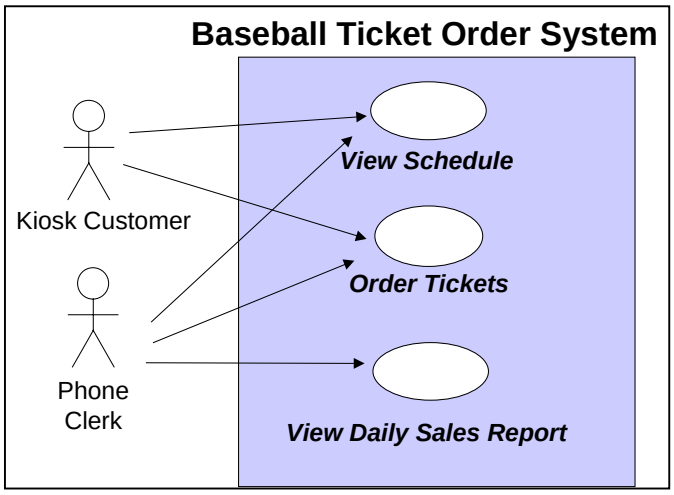

Figure 7: Actors with Overlapping Roles

The use case modeling notation provides a mechanism, **actor generalization**, for explicitly recognizing the commonality of actor roles. Figure 9 shows how the use case model can be redrawn with actor generalization to simplify the relationships between actors and use cases. This model says that *a Kiosk Customer* is a kind of *Ticketer* and that a *Phone Clerk* is a kind of a *Ticketer*. Any *Ticketer* may view a schedule or order tickets. A *Phone Clerk* (but not a *Kiosk Customer*) may additionally view a sales report.

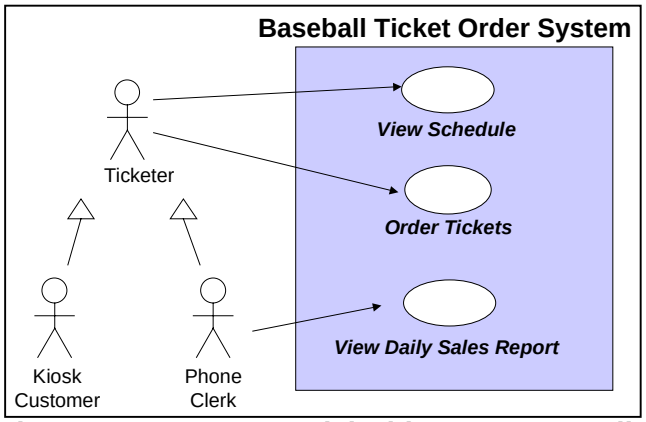

Figure 8: Use Case Model with Actor Generalization

Note that it would not have been correct to simply model *Phone Clerk* as a specialization of *Kiosk Customer*, in place of *Ticketer*. While the actor-to-use case relationships would be correct, the actor-to-actor relationship is semantically unsound: A *Phone Clerk* is **not** a kind of *Kiosk Customer*. (I saw that example in a recent use case book, which modeled a *Sales Rep* actor as a subclass of a *Customer* actor, in order to inherit the overlapping use case relationships.)

Problem #6: The use case specifications are too long.

Symptom: A use case specification goes on for pages.

Cure: The granularity of the use case may be too coarse. *Example:* **Use Schedule** (a use case that includes everything any user might want to do with a schedule) is too broad. More narrowly defined, specific use cases (such as **View Schedule** and **Create Schedule**) tend to be shorter and easier to understand.

Alternately, the granularity of the steps in the use case may be too fine. The steps may be too detailed or include purely internal processing (implementation). Rewrite them to focus on the essential interaction.

Problem #7: The use case specifications are confusing.

Symptom: The use case lacks context; it doesn't "tell a story."

Cure: Include a *Context* field in your use case specification template to describe the set of circumstances in which the use case is relevant. Make sure that the *Context* field puts each use case in perspective, with respect to the "big picture" (the next outermost scope). Don't just use it to summarize the use case.

Symptom: The steps in the normal flow look like a computer program.

Cure: Rewrite the steps to focus on a set of essential interactions between an actor and the system, resulting in the accomplishment of the actor's goal.

- Break out conditional behavior ("If...") into separately described alternate flows, leaving the normal flow shorter and easier to understand.
- Use case steps are not particularly effective for describing non-trivial algorithms, with lots of branching and looping. Use other, more effective techniques to describe complex algorithms (e.g., decision table, decision tree, or pseudocode).
- Make sure that the steps don't specify implementation. Focus on the external interactions. Consider expressing some of the behavior as "rules," rather than algorithms.

Problem #8: The use case doesn't correctly describe functional entitlement.

Symptom: The associations between actors and use cases doesn't correctly or fully describe who can do what with the system. This problem seems to occur for two reasons:

- The use case modelers were trying to be "object oriented," by making fat use cases that include all possible actions that might be performed on a business object. (I call these "CRUD use cases," since they often contain flows for creating, reading, updating, and deleting the object.) These use cases often have names that include the words "maintain," "manage," or "process."
- The use case modelers were trying to match up use cases to user interface screen. Faced with a view screen, that could also be edited (by a user with the right authority), they combined viewing and updating into a single use case that relates to the single screen design.

Example: Figure 10 shows a use case **Process Game Schedule**, that describes everything that any actor might want to do with a game schedule. Its specification has a "normal flow" for viewing the schedule, and alternate flows for updating the schedule. The *Kiosk Customer* actor may use the normal flow, but cannot use the alternate flow. Only the *Schedule Administrator* is functionally entitled to perform the schedule update.

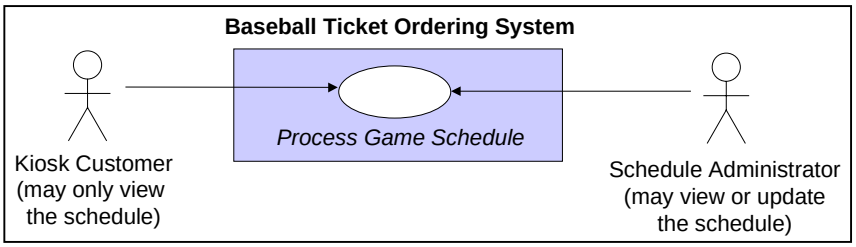

Figure 9: Confusing Functional Entitlement

Cure: Make sure that each actor associated with a use case is completely entitled to perform it. If an actor is only functionally entitled to part of the use case, the use case should be split. *Example:* The **Process Game Schedule** use case should be split into two: **View Game Schedule** and **Update Game Schedule**, as shown in Figure 11. Now it is clear, at a glance, that the *Kiosk Customer* may view, but not update, a schedule.

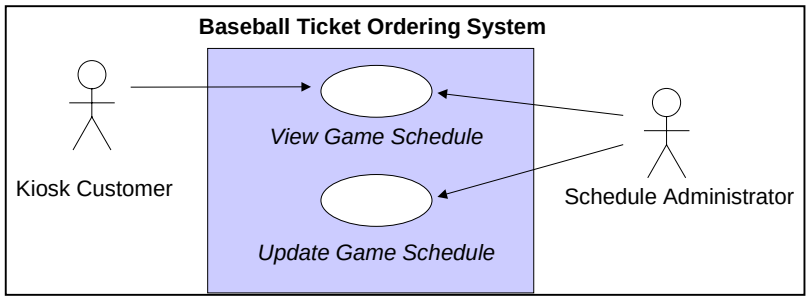

Figure 10: Use Cases with Correct Functional Entitlement

Problem #9: The customer doesn't understand the use cases.

Symptom: The customer doesn't know anything at all about use cases, but has been given a use case-based requirements document for review or approval. (Of course it's best when customers/end users have been included in the use case development. However, the person who reviews or approves the requirements may not have been involved in developing the use cases.)

Cure: Teach them just enough to understand.

- Put a short (1-2 page) explanation of use cases in the use case document, as a preface or appendix. The explanation should include a key to reading the model and specifications, and a simple example.
- Lead a short training session when use case document is distributed for review.
- Think long and hard about using <<***uses***>> and <<***extends***>> relationships in the use case model. They are a modeling convenience, but are not at all intuitive to the inexperienced reviewer.

Symptom: The use cases don't tell a story.

Cure: Add information to tell the story:

- Include a *Context* section in the use case template.

- Add an overview section that provides context to a set of related use cases (e.g., a package), and use this section to "tell the story."
- Include other kinds of models as needed. Often, a single use case will result in a state change to a major domain object, but the use case model alone won't tell the story of how the object changes state across many use cases over time. A state model (state transition diagram) of a major domain object may be an excellent way to show how several related use cases fit together over time.

Symptom: The use case organization doesn't match the way the customer thinks of the problem.

Cure: Determine what strategy for organizing the use cases makes the most sense to the customer. Listen to how the customer describes the business.

- How to partition the use cases into packages: Break out the use cases by major roles/actors, or by major events in the customer's business. *Example:* The customer talks a lot about "Spring Setup" -- when they put the new game schedule, stadium section definitions, and ticket prices into the system. Even if that's not the way the system developers think about the system, that's the package organization that makes sense to the customer.
- How to order use cases within a package: Order the use cases "chronologically," to describe a story of system use over time. Don't order the use cases alphabetically!

Symptom: Use case is written in "computerese."

Cure: Watch out for computer slang that is not part of the customer's vocabulary. *Example:* Say "The system displays the result screen," rather than "The result screen is invoked."

Symptom: The customer just hates the use cases.

Cure: Deliver what the customer wants. This doesn't mean that use cases can't be used as a requirements elicitation technique, if use cases are really the right technique for the job. But they might not be a primary delivered work product. *Example:* One customer has its own requirements document template, and it's not use-case based. But the system is highly operational in nature, and we feel that use cases are the best approach for eliciting and modeling the requirements. We perform the use-case based analysis, and then write the requirements in the format that the customer wants, based on what we learned in that analysis. The use cases might be included in an appendix to the requirements document, or they might not be a deliverable at all.

Problem #10. The use cases are never finished.

Symptom: Use cases have to change every time the user interface changes.

Cure: *Loosely couple* the user interface details and use case interactions. The user interface design is likely to change, and you don't want your system requirements to be dependent on design. (The dependency goes the other way -- the user interface design must satisfy the use case requirements.) A little coupling is okay -- "low fidelity" pictures of the user interface can aid understanding of the use case. But don't overly tie the fundamental interactions to the UI mechanisms (which are more likely to change). In the flows, focus on the essentials of what the actor does (e.g., "selects a game," "submits a request") rather than how the interaction is done (e.g., "double-click on the Submit button").

Specify use case "triggering" events as preconditions (e.g., "user has selected a game, and requested to order tickets"), rather than screen navigation details. Keep the screen navigation information in a (separate) user interface design document, not in the use case model.

Symptom: The use cases require change every time the design changes.

Cure: The easy answer is "Don't put design in your use cases." That's generally good advice when the use cases are at a computer system scope. The use cases should record what the system must do, not the design/implementation details. Make sure that your use case steps are not unnecessarily low level; that is, they should completely specify what they system must do, but no more than that. Put design information discovered during analysis into a separate Design Guidance document.

When use cases are defined at a subsystem scope, changes in system-level design (e.g., how functionality is partitioned between subsystems) can affect the subsystem requirements. Until the system design is stable (and explicitly documented), the subsystem requirements, including the use cases, will not stabilize.

Symptom: There are so many possible alternate cases!

Cure: Watch out for "analysis paralysis." There is a point at which the requirements are adequately specified, and further analysis and specification does not add quality. Cover the "80%" cases; do your best on the rest within the allocated budget of time and money.

Symptom: The requirements are just unknown.

Cure: Use cases have a simple, informal, and accessible format. This may lead to the deceptive conclusion that developing use cases is easy. However, the simplicity of the format does not mean that the requirements analysis process is any less critical or any easier. *Use cases are a mechanism for defining and documenting operational requirements, not magic.*

Conclusions

The pitfalls and problems described in this paper are not an indictment of use cases, but rather problems in the *application* of use cases by inexperienced practitioners. In our experience, most use case development teams include inexperienced members. Use cases may be a new technique to the organization, and are being used by the development team for the first time. Even when the analysts or system developers have experience with use cases, other team members may not. The ideal use case team includes customers, end users, and/or domain experts. In most cases, these team members will have no prior experience with use cases. The simplicity of the use case modeling notation and natural-language specifications make use cases extremely accessible to such team members. They may fully participate in the use case modeling and specification, but are likely to encounter the pitfalls described in this paper.

One suggestion for teams in which some or all of the members are new to use cases is perform periodic informal "in-progress" reviews of the use case models and specifications, in order to catch problems early in the development, and to educate the team members. Of course, a formal review or inspection of a finished use case document is also appropriate. The reviews can be made more effective by the use of a checklist to help identify these common problems. An example of such a checklist is available by email on request from the author.