Top Ten Ways Project Teams Misuse Use Cases -- and How to Correct Them

Part I: Content and Style Issues

by **Ellen Gottesdiener**
Principal
EBG Consulting

Use cases are a wonderful and powerful way to define the behavioral requirements of your software. They have evolved in style and form over the past decade and are now used to define user requirements by many requirements analysts, developers, and business experts involved in software projects -- both object-oriented and otherwise. But if you don't fully understand the ins and outs of use cases, it's easy to misuse them or make mistakes that can unintentionally turn them into "abuse cases."

In my work on software projects, I've facilitated numerous requirements workshops and have probably encountered every kind of error people can make in writing use cases. In this two-part series, I will present a view of how use cases can go awry and discuss ways to prevent this from happening. In this first article, we'll begin by defining use cases and their purpose and then identify ten "misguided guidelines" project teams often apply when they actually create use cases. And finally, we'll take a closer look at the first six of those "guidelines" -- which relate to the content and style of use cases -- and explore ways to correct them. Next month, Part II will address the last four misguided guidelines: the most common mistakes teams make when modeling use cases.

What Use Is a Use Case, Anyway?

A *use case* is a textual or diagrammatic (or both) description of both the major functions that the system will perform for external Actors, and also the goals that the system achieves for those Actors along the way. Use cases can be represented with text, a diagram or through both formats. Use-case text can contain different pieces of information, but at a minimum it will include names and a basic course of

action. Exception conditions and variation paths are also included in detailed use-case descriptions.

Scenarios describe typical uses of the system as narratives or stories; each narrative can be a few sentences or paragraphs. Scenarios are "played out" in the context of a path through a use case. You can think of a use case as an abstraction of a set of related scenarios.

Ten Misguided Guidelines Teams Follow for Use Cases

Now that we've seen what an ideal use case is supposed to be and do, let's see what happens when people actually try to create use cases. Below is my lighthearted translation of the "misguided guidelines" I've observed people following during my years as a project participant and consultant.

1. Don't bother with any other requirements representations.
   (Use cases are the only requirements model you'll need!)
2. Stump readers about the goal of your use case.
   (Name use cases obtusely using vague verbs such as *do* or *process*. If you can stump readers about the goal of a use case, then whatever you implement will be fine!)
3. Be ambiguous about the scope of your use cases.
   (There will be scope creep anyway, so you can refactor your use cases later. Your users will keep changing their minds, so why bother nailing things down?)
4. Include nonfunctional requirements and user interface details in your use-case text.
   (Not only will this give you a chance to sharpen your technical skills, but also it will make end users dependent on you to explain how things "really work.")
5. Use lots of extends and includes in your initial use-case diagrams.
   (This allows you to decompose use cases into itty bitty units of work. After all, these are part of the UML use-case notation, so aren't you supposed to use them?)
6. Don't be concerned with defining business rules.
   (Even if they come up as you elicit and analyze use cases, you'll probably remember some of them when you design and code. If you must, throw a few business rules into the use case text. You can always make up the rest when you code and test.)
7. Don't involve subject matter experts in creating, reviewing, or verifying use cases.
   (They'll only raise questions!)
8. If you involve users at all in use case definition, just "do it."
   (Why bother to prepare for meetings with the users? It just creates a bunch of paperwork, and they keep changing their minds all the time, anyway.)

9. Write your first and only use case draft in excruciating detail.
   (Why bother iterating with end users when they don't even know what they want, and they only want you to show them meaty stuff, anyway!)
10. Don't validate or verify your use cases.
    (That will only cause you to make revisions and do more rework, and it will give you change control problems during requirements gathering. So forget about it!)

If you recognize yourself in any of these "guidelines," take heart. The reason I know them so well is that I've made most of these mistakes myself. Pausing to examine your own mistakes is a wonderful way to learn, so now I'll share some of my experiences with use cases.

Correcting Misguided Content and Style Guidelines

As I've noted, the first six misguided guidelines relate to content and style issues, which we'll examine in this article, starting with:

1. Don't Bother with Any Other Requirements Representations

Because use cases are powerful and familiar software engineering tools, many teams mistakenly believe they can employ use cases alone to define user requirements.[1] But experience shows that use cases are often insufficient and in some cases inappropriate for this purpose.

Why? From the point of view of a person or system interacting with your software, a use case nicely describes an aspect of its behavior. But no single user requirements model can fully express all of the software's functional requirements: its behavior, structure, dynamics, and control mechanisms. Figure 1 illustrates how these requirements translate into four interrelated views.

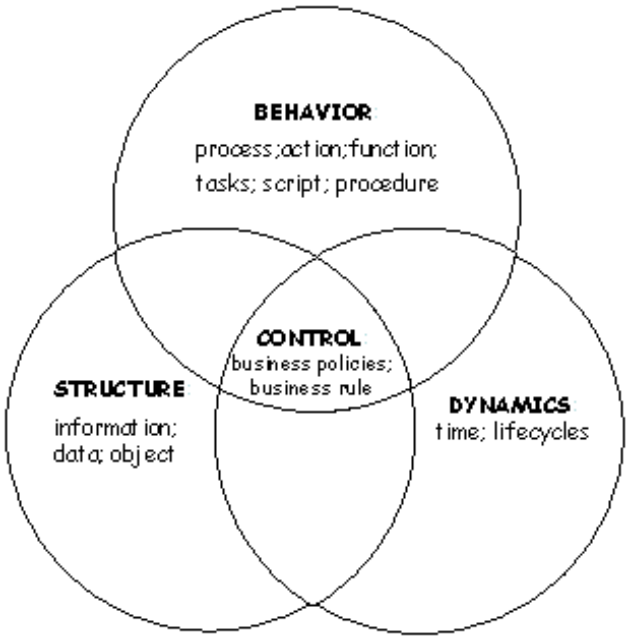

Figure 1: Four Views of System Requirements. *Employ use cases to define software*

behavior but other models to describe user requirements.

These views provide complementary mechanisms for analyzing your business domain and modeling it accurately and completely. Suppose, for example, that you're creating a product-ordering application. If you want to represent this domain in terms of use cases, then you might propose a use case such as "Place Order" to capture the flow of the ordering process. This use case might adequately describe this *behavior* of your system, but it would miss related structural, dynamic, and control elements. The *structural* view deals with attributes of the order, the order placer, and the customer. The *control* view encompasses rules for order placement, invoicing, billing, and back ordering. In describing the *dynamics* of placing an order, you might want to specify the allowable states of the order and actions that occur within those states.

In short, using multiple views gives you a richer context for eliciting user requirements. It also aligns with an important principle of requirements engineering: separation of concerns. Each model describes a specific aspect of your software and omits extraneous information. This means, for example, that your use cases don't include details found in other models, such as data attributes and business rules. Instead, these related models -- whether defined with a diagram or text -- should be traced to your use cases.

One project I worked on impressed me with how important it is to separate concerns. In our initial draft, we wove business rules into the use-case text, along with occasional lists of data attributes. But knowing that this information was sprinkled across multiple use cases, we removed it to other models. We made a list of the business rules in English, and we created a visual domain model that contained a logical data model and an analysis class model. We expected that the requirements would change as we worked and wanted the ability to quickly assess the impact of those changes. The changes rippled beyond a single model, so we used our requirements management tool (Rational® RequisitePro®) to associate the models with each other.

As our users explored their requirements and details evolved, we easily managed the changes. We collected business rules, the data, and the analysis class model at the same time, yet separately, from the use cases. We deftly bounced between models in a requirements workshop, and ultimately we reached closure faster than we would have if we had left the use-case text loaded up with lots of information.

Another project team I worked with learned why relying solely on use cases is problematic. The team was using facilitated workshops to determine requirements for a financial project. The goal was to support plant managers in querying information using a variety of reporting and query rollups of summary data. To model the system, the team planners wanted to create and verify use cases and actors as their primary deliverables. But after careful analysis, we realized that use cases weren't a useful way to express the problem domain. For this purpose, a single use case such as "Query Plant Information" would be far too abstract and all-inclusive. So instead, we used a structural view (data model) and a control view (business rules) to define user requirements. We elicited

these models by starting with scenarios in the form of situations and questions that plant managers would need to ask. For that particular project, and for similar data querying systems, use cases would be minimally useful.

In general, it's best to let the business problem domain drive the selection of the best requirements models for representing functional needs. Table 1 provides examples of some business domains and appropriate requirements models. Often, you can create simplified versions of what might otherwise be complex models, such as statecharts and data models, in collaboration with business experts or users during a workshop.

Table 1: Example Business Domains and Appropriate Requirements Models

Business Domain	Primary View (see Figure 1)	Suggested Models
Operations, Administration, Inventory Management, Billing, and Ordering	Behavior	Use Cases, Scenarios, Actor Table and Maps, Domain Models, Event Table, Prototypes
Data Query and Analysis, Data Extraction, Ad hoc and Standard Reporting, Customer Reporting	Structure	Data Model, Scenarios, Business Rules
Workflow, Logistics, Demand Management, Contract Negotiation, and Procurement	Dynamics	Process Maps, Event Table, Statechart, Prototypes, Scenarios
Claim Adjudication, Welfare Eligibility, Mortgage Lending, Clinical Diagnosis	Control	Business Rules, Statecharts, Scenarios, Event Table

Use cases are especially appropriate for highly interactive (behavioral) systems involving end users. Embedded systems, intensively algorithmic systems, data access, and batch systems might start with use cases such as "Provision Line Card," "Compute Dividend," "Query Information," or "Refresh Application Files." However, these types of systems won't benefit from the writing of detailed use case text. Other requirements representations, such as functional hierarchies or precise specifications like Gilb's Planguage,[2] are more effective.

Overcoming single-model-itis (the temptation to use use cases alone) will increase the quality of your requirements, reduce rework, and save you time and money. It will also speed requirements development and uncover requirements defects. On one project, we laid out use-case steps on a wall,[3] listed business rules below the steps, and listed data attributes nearby on sticky notes. As we discussed use-case steps, we found missing business rules and attributes. Each was separately documented yet traced to the other steps. The result? The project experienced no defects resulting from requirements errors, which gave strong endorsement to our approach.

2. Stump Readers About the Goal of Your Use Case

To paraphrase Alistair Cockburn, the purpose of a use case is to fulfill an Actor's goal in interacting with the system.[4] As you review your list of use cases, be sure that the goal and the Actor (the person or thing that has the goal) are clear.

"Process Order" or "Do Inventory" are vague use-case names, leaving a lot of room for interpretation. What is the goal of "Process Order"? Is it to authorize the order? Find available products? Pack and ship the order? Some combination of these? A single use case can't describe all the actions that name might encompass.

The best way to generate use-case names is either by starting with Actors or by listing use cases and then immediately naming each use case's Initiating Actor. Well-named use cases often enable a business customer to easily infer who the Actor is. An unclear name, in contrast, provides few clues. What Actor initiates the use case named "Process Order," for example? An order taker? An inventory replenisher? A shipper?

The following guidelines will help you avoid this naming problem.

- Name your use cases using this format: verb + [qualified] object.
- Use active (not passive) verbs.
- Avoid vague verbs such as *do* or *process*.
- Avoid low-level, database-oriented verbs such as *create*, *read*, *update*, *delete* (known collectively by the acronym CRUD), *get*, or *insert*.
- The "object" part of the use-case name can be a noun (such as *inventory*) or a qualified noun (such as *in-stock inventory*).
- Make sure that the project Glossary defines each object in the use-case name.
- Add each object to the domain model (as a class, entity, or attribute).
- Elicit Actors and use case concurrently, associating one with the other as you name each.

One project I know of had sixty-eight use cases because team members created a use case for each database event. But it doesn't make sense to describe database events this way. Use cases are designed to model Actor interactions with your software. Human Actors don't interact with the system in order to CRUD rows in their databases. They don't think in terms of rows and databases; they may not know what a database looks like internally, nor should they have to know that. Rather, Actors think in terms of higher-level goals, such as finding out the discount to give a particular customer; and these goals, in turn, serve business objectives.

Although goals (use cases) can be related, don't make the mistake of blending them together. For example, the use cases "Place Order,"

"Replenish Stock," "Locate Distributors," and "Ship Order" are related, but each is a distinct use case. If you understand their interdependencies,[5] then it will be easier to prioritize them and to plan increments with customers. It will also give you built-in flexibility if you need to trim functionality for a given release. But don't make the mistake of thinking of these cases as one big use case.

To help project teams create good use-case names in requirements workshops, I give participants a "cheat sheet" of good verbs to use (see Table 2). I divide the list into *informative* use cases (those that give information to the Actor) and *performative* use cases (those that execute a business transaction to deliver value to the customer or that change the state of data in the system).

Table 2: Example Verbs to Use in Use-Case Names

Informative Use Cases	Performative Use Cases
Analyze	Achieve
Discover	Allow
Find	Arrange
Identify	Change
Inform	Classify
Monitor	Define
Notify	Deliver
Query	Design
Request	Ensure
Search	Establish
Select	Evaluate
State	Issue
View	Make
	Perform
	Provide
	Replenish
	Request
	Set up
	Specify

This list can help you to arrive efficiently at a first-cut list of use cases without having to clarify their meaning. In one requirements workshop in which use cases were our primary deliverable, I started with a one-minute definition of the term "use case." Next, I handed out the cheat sheet, and together we named several use cases for the project. Then, while the developers and analyst watched, the three business experts present were able to generate more than fifty use-case names for three related subsystems -- in only twelve minutes! We spent another ten minutes or so clarifying, collapsing, and adding use cases to arrive at a first cut list of about fourteen use cases per subsystem. The participants went on to practice writing a one-paragraph description of a sample use case, and from there we iterated through the process of detailing each use case, mapping out dependencies, and packaging the use cases for prioritization and release planning.

3. Be Ambiguous About the Scope of Your Use Cases

Use-case scope mistakes are typically of two sorts: Either the use case does not address a single Actor goal, or the use case does not fall within your project's scope and should never have been detailed in the first place. Both types of mistakes waste a lot of time and energy. If you don't scope your use cases appropriately, then development becomes unnecessarily complex, and iterative and incremental development becomes a major chore. If you don't frame each use case clearly, then it's hard to know when a use case starts and ends.

To avoid confusion, remove out-of-scope use cases by naming them well (see the preceding section) and ensuring each belongs in the system's scope. Several other models can help, including the context diagram and event table. All surviving use cases should be checked to ensure that each addresses one or more of the business goals defined in your Vision or Charter.

To keep a single use case in scope -- and thereby address only one Actor goal -- constrain each use case with its triggering event and necessary event response. *Events* are what cause Actors to initiate use cases. When the *event response* is achieved, the use case is finished.

Events for scoping use cases come in two flavors: business and temporal. *Business* events are high-level occurrences that happen at unpredictable times. Although you can estimate, for example, how many book requests or product searches might occur, you can't specifically say *when* they will occur or *how often*.

Assign names to business events using a "subject + verb + object" format: for example, "customer requests book." In this example, one event response might be that book information is provided to the customer. As you might guess, the subject part of the business event turns out to be an Initiating Actor, and the verb part gives you clues for naming one or more use cases.

Temporal events, on the other hand, are entirely predictable occurrences, driven by a specific time on the clock. You know *exactly* (i.e., the month, day, year, hour, or minute) when the use case needs to replenish inventory levels, publish the schedule, post bills, or produce 1099s. Temporal events are driven by the clock and should be named using the format "time to <verb + object>." The initiating actor for these temporal events will be "Clock" or a psuedo actor name you choose such as Inventory Controller or Scheduler Manager. Event responses to temporal events can be what McMenamin and Palmer[6] call "custodial" -- for example, cleaning up data inside the system by refreshing information -- and they can generate tangible artifacts to actors, as business events often do.

It's useful to define events and their responses in your *use-case template* -- a standard format for documenting use cases. Templates usually are headed by high-level information about the use case. A business or temporal event is not itself a use case; rather, it corresponds to the "trigger" in most use-case templates, and the event response corresponds

to the "success outcome" in your use-case header.

Defining events can also help you eliminate use cases that don't belong in your project's scope. Let's look at a few ways.

- Events can be your starting point in defining requirements. In fact, an event name is very similar to a use-case name, simplifying the transition from scope to use cases (see Figure 2). To add vigor to your scoping activity, it's a good plan to use a context diagram or context-level use case to describe events and event responses.
- Drawing a context diagram while simultaneously naming business and temporal events allows everyone to "see" the system's scope. On the context diagram, business events are in-flows to the central bubble (or oval), and event responses are shown as out-flows (the system's response to the external environment). Temporal events can be out-flows, and sometimes also in-flows, when the temporal event requires feeds from external Actors.
- In an hour or less, you can create an *event table* (a table with one column for events and another for the corresponding event responses) along with a context diagram. It's an hour well spent, because it helps you avoid specifying use cases that don't belong.
- If your team is eager to jump into naming use cases, consider taking a brief detour to the event table and context diagram. By taking five to fifteen minutes to refresh everyone on the project's scope, you will likely uncover missing events or extraneous use cases, saving significant rework later on.

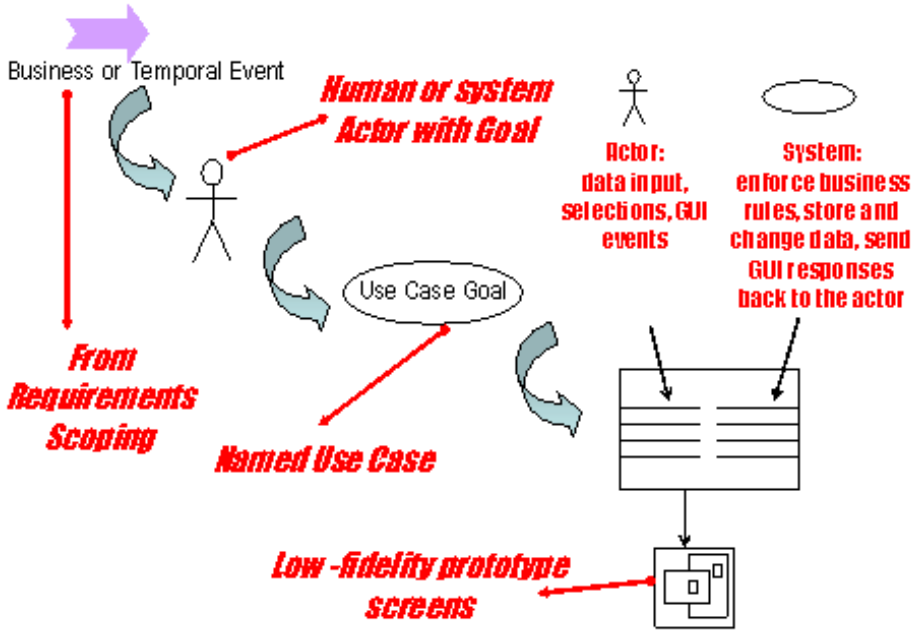

Figure 2: Events Can Help You Frame Use Cases Precisely

4. Include Nonfunctional Requirements and User Interface Details in Your Use-Case Text

A common mistake teams make with use cases is to incorporate nonfunctional requirements, such as response times and throughput information, and user interface or prototype details such as widget manipulation and references to windows, buttons, and icons. Although use cases are effective tools for eliciting nonfunctional requirements and for envisioning the user interface, you shouldn't insert that information into your use-case text; instead, you should associate that information to the relevant use cases. For example, you can define nonfunctional requirements in a nonfunctional requirements specification or supplementary specification document that traces each nonfunctional requirement to its corresponding use case. The prototype sketches, drawings, or screens should be also stored separately, and traced to the use cases they envision.

Nonfunctional requirements include quality attributes (such as performance, throughput, and usability), system requirements (such as security and recoverability), and system constraints (such as database and language). These nonfunctional requirements drive architectural decisions, govern user satisfaction with your final product, and provide competitive advantage in commercial software products.

User quality attributes constrain functional requirements. Use cases, and any other user requirements model describing software functionality, portray the "doing" part of requirements. Their associated nonfunctional requirements describe the "being" part of software. You should strive to separate the *doing* from the *being* but also relate the two.

As you begin to specify the functionality needed to achieve a use case, you can uncover some of your nonfunctional requirements, such as response time, throughput, and usability. To do that, ask good questions of users -- or surrogate (stand-in) users -- while eliciting use cases:

- How many users does this Actor represent?
- What is the maximum response time acceptable for <use case>?
- How many <object part of use case> will you need to <verb part of use case name> each day, hour, or week?
- Are there periods in the year when you will see higher volume?
- Will experienced and new users need to learn to use this functionality differently?

The answers will help you begin to nail down the nonfunctional requirements for your use case.

Other nonfunctional requirements -- such as backup, recovery, security, and audits -- relate to multiple use cases. Often you need define them only once, and they don't belong in your use-case text. Separating them will help your project architects get a comprehensive overview of the technical issues that will drive important design considerations.

Prototypes describe requirements as viewed by direct users, or actors.

Before you code anything, try eliciting and testing use cases by using simple screen navigation maps or mockup dialogs posted in sequence on a wall. These low-fidelity prototypes also help you manage user and customer expectations about the system's look and feel without locking anyone into specific interface designs.

Though it's tempting to incorporate GUI references into use-case text, you shouldn't fall into this trap. It creates design expectations that may prove erroneous or unworkable as you iterate through Elaboration and Construction. In one project I know of, the team had to rework their use-case text when they embedded specific GUI references to an environment (Java Swing) that changed (to XLS) shortly after they had drafted their use case. The use-case text should apply regardless of implementation environment.

In sum, let your use cases do what use cases do well: describe Actor-system interactions. To describe constraints and quality attributes for those interactions, define and trace nonfunctional requirements apart from your use cases. To describe how the software will look and feel, use a prototype.

5. Use Lots of Extends and Includes in Your Initial Use-Case Diagrams

Extensions and includes are among the most confusing aspects of the use-case diagram. Overzealous attempts to use the notation -- just because it's there -- can lead to analysis paralysis.

In practice, <<include>> use cases aren't revealed until the second or third iteration through all the use cases. On one project I facilitated, we iterated first through all the "Happy Paths" (normal scenarios in which all goes well) and then all the "Unhappy Paths" (assuming errors, exceptions, and infrequently occurring scenarios) over the course of several days. The number of use cases expanded and contracted as we explored the breadth and depth of the project. We began with fifteen use cases, went down to fourteen, and finally settled with twelve. *Included* use cases became apparent as we iterated through the set, finding patterns of behavior that could be partitioned out and reused by multiple use cases. Jumping to <<>> use cases too soon leads to the trap of a functional decomposition mindset, eradicating the advantages that Actor- and goal-driven use cases provide.

Extensions inside use-case text add complexity because they often address important errors or exception conditions -- in other words, business rule violations. Occasionally, a set of steps that handles similar extensions turns out to be an included use case. To save time in identifying these patterns, you should define business rules explicitly.

Although the use-case diagram with includes and extensions semantics might be useful for certain complex use cases, it is more productive to spend your time specifying use-case text, visualizing relationships within and between use cases, and using multiple requirements models (sound familiar?). To more easily visualize each use case, ask users to lay out

each step on a wall and then step through the use case. At each step, ask them questions designed to uncover attributes, business rules, and elements that might appear on a user prototype.

6. Don't Be Concerned with Defining Business Rules.

This guideline is based on a serious misconception: Business rules should be *at the heart* of your use cases, providing the controls, reasoning, guidance, and decisions behind the processes the use case describes. Business rules exist to enforce higher-level business policies, which in turn serve business goals, objectives, and tactics. If you don't explicitly define and separate your business rules, they will almost surely end up wrong, missing, or difficult to change. Many post-implementation defects relate to problems with business rules.

Business rules are owned and known (or need to be known) by business experts and/or the product and marketing managers who represent them. Technical people have no business trying to guess at business rules unless they are well versed in the business and are authorized to define the rules.

For this reason, early in requirements efforts, I request that a business executive or expert assume the role of project "Rule Czar." This means taking responsibility for defining the rules and noting where they apply. As you elicit use cases, many questions will arise about business rules, and the Rule Czar's job is to help the group reach closure on such questions.

To find out whether business rules are lurking below the surface of your use cases, listen and look for certain verb clues in the text you have written:

- evaluate
- determine
- assess
- verify
- validate
- classify
- decide
- compare
- diagnose
- match
- conclude
- should (as part of verb phrase)

Once you hear these verbs, ask probing questions of your users or customers to uncover the business rules that must be enforced to take the action the verb suggests (e.g., verify, decide, etc.) in the use-case description.

To help you specify business rules precisely, you can use *business rule templates*, which give you a structured format for writing business rules in natural language. This will help you tease loosely written business rules into atomic business rules. And as you do so, you'll find missing elements from other models, such as the domain model and use-case steps.

There is no agreed-upon taxonomy for business rules, nor does there need to be. Table 3 shows some examples. Each project is unique, so you must select or invent a template that works best for your domain. Be sure to define each term in your business rules in your *Glossary*. Terms are the building blocks of all your business rules and are used throughout your use cases, so you should nail down their meanings as soon as you can.

Table 3: Sample Business Rules Templates

Category	Templates	Examples
Term (list in Glossary)	[property] <noun/business term> is defined as <text definition>	A manager is defined as a person to whom two or more people report directly.
Fact	Each <noun/business term> must|may <verb or verb phrase> one and only one|one or more <noun/business term> [<prepositional phrase>] <noun/business term1> may|must <verb or verb phrase> <noun/business term2> <noun/business term1> has a property of <noun/business term2>	Each buyer must assign one and only one discount to an order. Line items must contain the quantity requested. "Web customers" has a property of "userid."
Constraint	<[qualified] noun/business term> must be true for <condition> or <condition [property] <noun/business term> must not/cannot <verb phrase> <constant or non-verb phrase>	The active ingredient for a finished product must be listed first on the package. Total-sale must not exceed $100. An underage customer cannot purchase alcoholic beverages from liquor stores.

Derivation	<noun/business term> is calculated as <arithmetic expression>	Total excess material is calculated as (total volume input minus total amount used).
Action Enabler (also known as ECA rules: event, condition, action)	when <condition is true>, then <action> if <condition1> [and <condition2>...] then <action>	When claim arrives after cancellation date, then issue rejection letter. If preferred customer and backordered item, then offer 10 percent discount.
Inference	If <condition1 [true]> [and condition2...] then <conclusion>	If customer submitted expired credit card, then credit is suspicious.

Be sure to separate business rules from your use cases but still relate them. On one project we included business rules in our use cases from the start, and by the second iteration we found ourselves diving into multiple use cases to change and add business rules. We corrected course by deleting the business rules from the use-case text and turning them into a distinct requirements artifact; then we traced use cases to business rules and vice versa, using traceability matrices. This taught me a valuable lesson: Agility is facilitated not only by simplicity but also by separation.

Write your use case with no business rules embedded in the text, but reference them in a separate document and trace them to the use case that must enforce them.

Until Next Time

In this article we've come a bit more than halfway through the list of misguided guidelines. Next month we'll look at the last four, which are pitfalls to watch for in the process itself. Until then, remember that although use-case mistakes are common, it is worth the effort to correct them. Use cases can really work for you if you don't misuse them!

Acknowledgments

I would like to thank the following reviewers for their helpful comments and suggestions: Alistair Cockburn, Gary Evans, Susan Lilly, Bill Nazzaro, Paul Reed, Debra Schratz, Karl Wiegers, and Rebecca Wirfs-Brock.

References

Alistair Cockburn, "Use Cases, Ten Years Later." *Software Testing and Quality Engineering Magazine* (STQE), Vol. 4, No.2, March/April 2002, pp. 37-40.

Alistair Cockburn, *Writing Effective Use Cases*. Addison-Wesley, 2000.

Alistair Cockburn, "Using Goal-Based Use Cases." *Journal of Object-Oriented Programming*, November/December 1997, pp. 56-62.

Martin Fowler, "Use and Abuse Cases." *Distributed Computing*, April 1998.

Tom Gilb, *Competitive Engineering: A Handbook for Systems and Software Engineering Management Using Planguage*. Addison-Wesley (forthcoming, 2002).

Ellen Gottesdiener, *Requirements by Collaboration: Workshops for Defining Needs*. Addison-Wesley, 2002.

Ellen Gottesdiener, "Collaborate for Quality: Using Collaborative Workshops to Determine Requirements." *Software Testing and Quality Engineering*, March/April 2001, Vol 3, No. 2.

Ellen. Gottesdiener, *Requirements Modeling with Use Cases and Business Rules* (course materials, EBG Consulting, Inc.), 2002.

Daryl Kulak and Eamonn Guiney, *Use Cases: Requirements in Context*. Addison-Wesley, 2000.

Susan Lilly, "How to Avoid Use Case Pitfalls." *Software Development Magazine*, January 2000.

Stephen M. McMenamin and John F. Palmer, *Essential Systems Analysis*. Yourdon Press, 1994.

Rebecca Wirfs-Brock and Alan McKean, "The Art of Writing Use Cases." Tutorial for OOPSLA Conference, 2001. See http://www.wirfs-brock.com/pages/resources.html

---

Notes

[1] "Requirements" define the operational capabilities of a system or process that must exist to satisfy a legitimate business need. The generic term "requirements" covers both functional (functionality users expect) and nonfunctional requirements (quality attributes of the software such as performance, system needs such as security and archiving, and technical constraints such as language and database). Functional requirements evolve from user requirements-tasks that users need to achieve with the software.

[2] Plangauge is a specification language developed by Tom Gilb. For more information, see: www.result-planning.com and Tom Gilb, *Competitive Engineering: A Handbook for Systems and Software Engineering Management Using Planguage*. Addison-Wesley (forthcoming, 2002).

[3] For more about this technique, see "Specifying Requirements with a Wall of Wonder" in the November issue of *The Rational Edge*.

[4] Alistair Cockburn, *Writing Effective Use Cases*. Addison-Wesley, 2000.

[5] Two useful ways to understand use-case dependencies is to show how use cases execute in sequence with a use-case map (see http://www.ebgconsulting.com/publications.html, "Use Cases" section for an example) and by clearly defining pre- and post-conditions for each use case.

[6] Stephen M. McMenamin and John F. Palmer, *Essential Systems Analysis*. Yourdon Press, 1994.

For more information on the products or services discussed in this article, please click here and follow the instructions provided. Thank you!