

Managing Use-Case Details

by **Kurt Bittner**
General Manager
Process and Project Management Business Unit

Details, details! System designers seem to have a general fear of details when writing use cases, perhaps driven by the worry that clarity, the hallmark of a good use case, will be lost in a blizzard of details if they are not careful. Although this fear has some foundation in reality, unfortunately the typical response -- to omit all details from the use-case description -- results in use cases that lack specificity and real value.

As the old saying goes, "The devil is in the details," meaning that most of the real problems become apparent only when you get down to specifics. If we are to write effective use cases, then we must present details. So how can we overcome our fears about them?

The best strategy is to plunge ahead. All the details will be needed at some point, and you can move them to other artifacts later on. As Franklin Roosevelt said, "We have nothing to fear but fear itself." Sometimes fear of detail is just procrastination in disguise: We worry about documenting a behavior with too much detail before we even start writing anything and then find ourselves unable to get started. Instead, try to put aside your fears and dig into the specifics of the required behavior to describe exactly what the system will do. If you do not provide enough detail, then there *is* a real possibility of failure: The development team will have to re-discover the requirements even after they read your use-case description. You will have wasted everyone's time, including your own.

This article discusses use-case flow description and presents several strategies for managing details in these descriptions, including using glossaries and domain models, plus representing complex business rules and other special requirements. En route, a number of examples are used to illustrate various aspects of the problem. So without further

introduction, let's get started.

How Much Is Just Enough?

Or, put another way, "How can we write use cases without 'losing the forest for the trees'"?

A use case needs to *unambiguously* describe the required behavior, no more and no less. If the system must respond in a certain way to a certain event, then you must say so in the use case. The trick is to express the behavior without dictating or constraining the design. For example, consider the following simple use case:

Example (from an Automated Teller System)

Use Case - Withdraw Cash - Main Flow

1. The use case starts when the customer inserts his bank card.
2. The system reads the card and requests the customer to enter a Personal Identification Number (PIN).
3. The system presents a menu of choices.
4. The customer indicates that he wishes to withdraw cash.
5. The system requests the amount to be dispensed, and the customer enters the amount.
6. The system dispenses the desired amount of cash, prints a receipt, and ejects the card.
7. The customer takes the cash, card, and receipt.
8. The use case ends.

Simple enough, right? Actually, too simple; all sorts of important details are missing: What information does the system read from the banking card? What does the system do to verify that the correct PIN has been entered? How does the system know that the customer has sufficient funds in his account? What information gets printed on the receipt?

Look what happens if we start including this information (in *italics*):

Example (from an Automated Teller System)

Use Case - Withdraw Cash - Main Flow

1. The use case starts when the customer inserts his bank card.
2. The system reads the *bank* card, *obtaining the bank number, the account number, and the Personal Identification Number (PIN) from the card. The system then* requests the customer to enter his PIN. *The PIN can be up to 6 digits in length and must not include any repeated digits.*
3. *The system compares the entered PIN to the PIN read from the card to determine if the PIN entered is valid.*
4. *If the PIN entered is valid, then* the system presents a menu of *transactions: Withdraw Cash, Deposit Funds, Transfer Funds, See Account Balances.*
5. The customer indicates that he wishes to withdraw cash.
6. The system requests the amount to be dispensed, and the customer enters the amount.
7. *The system checks to see if it has sufficient funds in its dispenser to satisfy the request.*
8. *The system ensures that the amount entered is a multiple of $5 and does not exceed $200.*
9. *The system contacts the customer's bank to determine if the amount requested can be withdrawn from the customer's bank account.*
10. *If the customer has sufficient funds on hand, then* the system dispenses the desired amount of cash.
11. *The system* prints a receipt *with the following information:*
    - *Date and time of transaction*
    - *Location of the ATM*
    - *Customer's bank number*
    - *Type of transaction*
    - *Amount of the transaction*
    - *Transaction identifier (for tracking within the inter-bank network)*
12. *The system* ejects the card.
13. The customer takes the cash, card, and receipt.
14. The use case ends.

Of course, the description gets quite a bit longer when we add details about what the system does and what information is captured. Some of

you are probably thinking that this is too much detail, but ask yourself this: If you were paying someone lots of money to develop the system, wouldn't you want to know exactly what the system was going to do? Also note that none of the "details" dictate how the system should be designed.

So, if it takes almost a page to describe one use case for a "simple" system like an ATM, you may be thinking, how many pages would a "real" system consume? And how can you keep the details from overwhelming the reader? We'll look at some strategies for managing details below.

Create a Glossary

Use cases often contain information that can be presented more effectively in other ways. Creating a glossary of terms is one way to present background information that can otherwise be distracting to the reader.

The *Withdraw Cash* use case, for example, includes a number of terms that need to be defined; we will highlight some of the key terms in boldface text in our description:

Example (from an Automated Teller System)

Use Case - Withdraw Cash - Main Flow

1. The use case starts when the **customer** inserts his bank card.
2. The system reads the ***bank* card**, obtaining the *bank number, the* ***account number*** *and the* ***Personal Identification Number (PIN)*** *from the card. The system then* requests the customer to enter his PIN. *The PIN can be up to 6 digits in length and must not include any repeated digits.*
3. *The system compares the entered PIN to the PIN read from the card to determine if the PIN entered is valid.*
4. *If the PIN entered is valid, then* the system presents a menu of *transactions: Withdraw Cash, Deposit Funds, Transfer Funds, See Account Balances.*
5. The customer indicates that he wishes to withdraw cash.
6. The system requests the amount to be dispensed, and the customer enters the amount.
7. *The system checks to see if it has sufficient funds in its dispenser to satisfy the request.*
8. *The system ensures that the amount entered is a multiple of $5 and does not exceed $200.*
9. *The system contacts the* ***customer's bank*** *to determine if the amount requested can be withdrawn from the customer's bank account.*

10. *If the customer has sufficient funds on hand, then* the system dispenses the desired amount of cash.
11. *The system* prints a **receipt** *with the following information:*
    - *Date and time of transaction*
    - *Location of the ATM*
    - *Customer's bank number*
    - *Type of transaction*
    - *Amount of the transaction*
    - *Transaction identifier (for tracking within the inter-bank network)*
12. *The system* ejects the card.
13. The customer takes the cash, card, and receipt.
14. The use case ends.

Although we could define these terms in the use case itself, there are good reasons not to:

- It would be distracting and get in the way of the flow of events description.
- Other use cases for the system probably use the same terms, so it is more efficient to define the terms once, in one place.

So instead, we can put the terms in a glossary. It's best to start creating a glossary as soon as special terms start popping up. This often happens while you are still discovering the use cases, so simply compile a list of terms to define later.

From the use case description above, we can begin creating a glossary:

Example Glossary for ATM System (partial)

Customer A person who holds one or more accounts at a member financial institution of the ATM inter-bank network.

Customer's bank The financial institution that issued the bank card, and at which the customer holds one or more accounts.

Account A service provided by a financial institution to maintain a customer's money or securities. Each account is assigned an account number. The financial institution is obligated to pay the customer, upon demand and adhering to the terms of the account agreement, a defined sum of money.

Bank card A physical identification device imprinted with magnetic information pertaining to the issuing financial institution:

- Bank number (assigned to the issuing financial institution by the government)
- Customer number (assigned by the issuing financial institution to the customer)
- Personal Identification Number (PIN), chosen by the customer at the time the card was issued

Personal Identification Number (PIN) An identification number, chosen by the **customer,** used in conjunction with the card for security purposes. The PIN can be up to 6 digits in length and must not include any repeated digits. To verify the identity of a customer, the ATM system requires the customer to enter her PIN; when the customer enters the same number as the PIN stored on the card, the system authenticates the customer's identity and allows her to proceed with account transactions.

Receipt A physical, printed record of the transaction(s). The receipt presents the following information:

- *Date and time of transaction*
- *Location of the ATM*
- *Bank number*
- *Type of transaction*
- *Amount of the transaction*
- *Transaction identifier (for tracking within the inter-bank network)*

Actually, the text contains additional terms we may want to include in this glossary: *dispenser, menu of transactions, amount to be dispensed,* and

so on. For now, however, we will use this evolving glossary to see how we can begin slimming down our use case. Note how we have moved some details from the use case to the glossary in the example shown below:

Example (from an Automated Teller System)

Use Case - Withdraw Cash - Main Flow

1. The use case starts when the **customer** inserts his bank card.
2. The system reads the **bank card** information.
3. The system compares the entered **PIN** to the PIN read from the card to determine if the PIN entered is valid.
4. If the PIN entered is valid, then the system presents a menu of transactions: Withdraw Cash, Deposit Funds, Transfer Funds, See Account Balances.
5. The customer indicates that he wishes to withdraw cash.
6. The system requests the amount to be dispensed, and the customer enters the amount.
7. The system checks to see if it has sufficient funds in its dispenser to satisfy the request.
8. The system ensures that the amount entered is a multiple of $5 and does not exceed $200.
9. The system contacts the **customer's bank** to determine if the amount requested can be withdrawn from the customer's bank account.
10. If the customer has sufficient funds on hand, then the system dispenses the desired amount of cash.
11. The system prints a **receipt** for the transaction.
12. The system ejects the card.
13. The customer takes the cash, card, and receipt.
14. The use case ends.

Now that we have the glossary, we don't have to define these terms again when we write the next use case. Sometimes, however, a glossary is not the only strategy we need to manage details.

Add a Domain Model for Interrelated Concepts

Although our simple ATM system does not illustrate the point very well

(since it deals with very simple information), often glossary terms are tightly related. Consider the following definitions for an online order entry system.

Order A contract between the company and a customer to provide a set of **products** to a particular **customer.** An order includes an order date, a shipped date, and order number, plus an **item** list and **product** information.

Line Item A number or alpha-numeric combination that specifies both the particular **product** and the quantity of that product being ordered. Appears on an **order.**

Customer Purchaser of the **items** on an order.

Product Commodity that is being sold to the **customer.** The product information on an **order** includes an identification number, description, and unit price.

Although it's perfectly acceptable to present this information in a glossary, as you may have noticed, there are a lot of cross-references in the definitions. That's a tip-off to the existence of structured relationships among the concepts.

Moreover, these structured relationships are actually evident *within* the entries: Each entry captures more information than just a simple definition of a term and would not be complete without reference to *other* terms. For information like this, which has structure or tight interrelationships, a *domain model* can be a useful presentation strategy.

A domain model provides a way to capture the relationships among concepts, as well as the structure of information within those concepts. Domain models are typically represented in diagrams (see Figure 1). Business object models[1] and enterprise data models (the parts that relate only to business terms) are types of domain models.

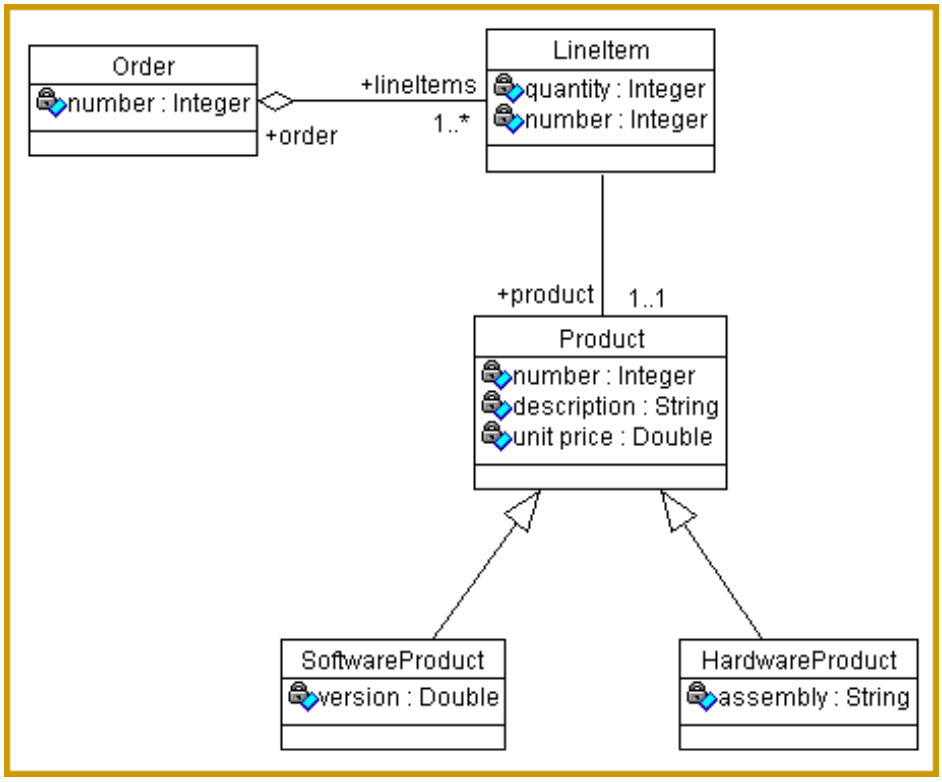

Figure 1: A Simple Domain Model Diagram

Use the Model for Definition, Not Design

A domain model does not represent the design of the system; it simply defines in precise terms a set of concepts used in the problem domain. The ability to visualize these concepts can help the team and various stakeholders agree on the definition of these concepts. Despite the fact that the domain model will eventually give rise to design elements, it is not wise to start capturing design information while working on the use cases. It may be tempting to say to yourself, "It's work I'll need to do eventually, so why not start now?" But in truth, it's far better to focus on doing one thing at a time and doing it well.

The purpose of the domain model is to clarify concepts and facilitate communication. If half the team starts diving deeply into system design, then you will lose sight of this goal, and the design will suffer. Before you can develop a really good design, you must understand the problem, including all the key concepts. Simple problems can quickly become more complex if you fail to focus on just one thing at a time.

Capture Simple Business Rules

Many use cases encompass business rules that relate to how information is validated. For example:

- Postal codes in addresses must be complete and correspond to valid codes.

- Product prices must be positive and end in whole dollar amounts.
- Customers can order only in-stock products.

These simple rules can often be captured in the domain model. You can specify minimum and maximum amounts as well as other validation criteria right along with the definition for the information itself. This relieves you from the tedium of describing every criterion and prevents the reader from getting lost in those details. Use cases excel at describing what happens and when -- i.e., real flows of events and activities. If you interrupt the description repeatedly to describe how data is validated, then your reader will lose a sense of the flow.

Other business rules relate to the way that work is performed or the way information is used or updated. Again, you can incorporate many of these rules into the domain model to simplify your work and make the activity flows easier to understand. More complex business rules, however, may require a different approach. See "Capturing Complex Business Rules" below.

Using Both a Glossary and Domain Model

If you use both a glossary and a domain model to manage details, it is important to decide where to define a term and then do it in only one place. If the concept is related to other concepts in well-defined ways (e.g., orders have items, which refer to products, etc.), then use the domain model. If the concept is simply a standalone term, then use the glossary. Establish clear guidelines on which artifact to use and when, and then apply them consistently.

It is also important to keep in mind that neither the glossary nor the domain model is intended to be a design document; they both exist only to clarify the requirements and use cases, not to start defining the system's internal structures. It is tempting to say, "Well, since we are doing a domain model, let's define all the things in the problem domain so that we'll have a complete model." That is all very well and good, but it's probably not what you're getting paid to do (which is probably to deliver a specific system). If the glossary and domain models help to clarify the problem domain and make it easier to describe what the system should do, then they have done their duty. To ask for more is to ask for trouble.

Capturing Other Information

Some of the information we classify as "detail" does not lend itself to either a glossary or a domain model. We will discuss strategies for capturing this information below.

Complex Business Rules

As we saw above, a *domain model* provides an excellent way to capture simple business rules, especially those that constrain relationships between things or specify the validation of information. These simple rules can be captured as requirements and traced to the use cases to which

they relate.

Other business rules, however, are requirements that constrain how the business itself works; they are independent of how the solution supports the business. These rules may require different responses. Here are some examples:

- A customer must be a member of the cooperative to make a purchase.
- A customer may have more than one outstanding order.
- A product may be sourced from more than one supplier; the product may have different prices depending upon the supplier.
- Customers whose bills go unpaid for more than 60 days will be referred to a collection agency.

The first three rules can be captured through relationships in the domain model. The last rule will need its own use case because it requires the system to make a decision and take some course of action.

Non-Functional Requirements

Use cases are great for representing functional system requirements: They describe how users interact with the system and what the system does in response. But many system requirements are *non-functional*: They state various conditions or constraints with which the system or its developers must comply. They may relate to *general* functionality, usability, reliability, performance, and supportability of the system. Teams often run into great difficulties when they try to apply use cases to handle these requirements.

There are several classes of these non-functional requirements:[2]

1. Non-functional requirements that apply to the system as a whole.

These may relate to overall quality and cost goals, or they may be general statements of characteristics that apply to the entire system. For example:

- The system shall be portable across Microsoft Windows and UNIX platforms (including Linux).
- The system shall have no more than 10 hours of scheduled downtime per year and must have no unscheduled downtime.
- The system shall have a mean time between failure (MTBF) of no less than 10,000 hours.
- The capitalized system development cost shall be no more than 10 percent per delivered unit.

These requirements cannot be traced to any particular use case and should simply be managed at the project level, even if they will have a

significant impact on the system's architecture. One *could* trace these requirements to *all* use cases, but this would quickly become burdensome. In addition, satisfying these requirements requires a coordinated solution; if you were to assign them to individual use cases, then different teams assigned to different use cases might pursue different strategies for them.

2. Non-functional requirements that are constraints on the solution.

These requirements dictate that specific technologies be used, or that specific algorithms or approaches be employed. They typically ensure that the new system will be compatible with the technologies of existing systems, or to make sure that certain approaches are followed to ensure consistent results. For example:

- All measurements reported by the system shall be provided in metric units, and all inputs to the system shall be made in metric units.
- The system shall utilize an X/Open XA-compliant transaction-processing monitor.
- The system shall access information using an ODBC-compliant database interface.
- The system shall report system management events using the SNMP standard.

In some instances, these constraints may apply only to a few use cases: reporting system management events or using a particular database interface standard, for example. If the requirement applies only to a specific use case or a small number of specific use cases, then trace it to the applicable use cases. Otherwise, leave it as a general requirement on the solution.

3. Non-functional requirements that relate only to a single or a few use cases.

Some non-functional requirements relate to specific use cases. These often convey additional qualities that the system must support when providing the functionality described in the use case. For example:

- Transaction *Y* must be completed in less than two seconds.
- The system must support a minimum of 200 concurrent instances of *use case X*.

These requirements should be traced directly to the use cases to which they apply.

As the use cases evolve, keep track of these links. When presenting a use case you will also need to present the non-functional requirements that apply to the use case. Typically, developers place these in a separate section of a use case report along with other special requirements.[3]

Special Requirements

Some supplementary requirements are hard to represent in the use case itself and yet are important to track to the use case. For example:

- Reliability requirements, such as "The system must be operational continuously, with less than four hours of scheduled down-time per year, and with no unscheduled down-time".
- Constraints on system design and implementation, such as "The system must conform to IEEE standard XXX," or "The system must be implemented in the Java programming language and must utilize J2EE," or "The system must run on a Microsoft Windows platform," or "The cost of the system must be no more than X per unit."
- Security requirements, such as "All access to system capabilities must be authorized."

These requirements do not belong in the use-case description because they really do not affect the flow of events,[4] but it is very important that they not be forgotten when the use cases are designed and implemented.

These requirements can either be documented in a separate section in the use-case description as "Special Requirements," or simply traced to the use case (if you're using a requirements management tool).

Different Details, Different Strategies

We have seen that use-case descriptions should be detailed; without the details, the use case cannot describe what the system will really do and will not enable developers to understand it. It is no secret, however, that details can sometimes get in the way of understanding. The strategies described above provide a number of ways to *manage* these details in a way that promotes better understanding. To decide which ones are best for your project, carefully assess the details you need to include. Briefly:

- Use a *glossary* to define simple concepts that have limited relationship to other concepts.
- Use a *domain model* in conjunction with the glossary if the concepts are interrelated to represent the structural relationships between the concepts. Include simple business rules in the domain model.
- Place non-functional and special requirements (reliability, design and implementation constraints, security) in a separate *"Special Requirements"* section within the use-case description or trace them to the appropriate use case.

As with any art, the right approach will probably involve blending these strategies in proportions guided by experience.

Want more information and advice on creating better use-case descriptions? See "UML Activity Diagrams: Versatile Roadmaps for Understanding System Behavior" in this issue of The Rational Edge.

---

[1] Jacobson et al., in *The Object Advantage*, use a business object model to capture the dynamics of the entire business process. In this context, the domain model is a subset of this business object model containing just the parts that identify key business entities and their relationships.

[2] This is not intended to be an exhaustive taxonomy of the types of non-functional requirements; it's just a characterization to help explore some of the typical problems encountered when mapping non-functional requirements to use cases.

[3] Note that the important thing is not what you call these "special" requirements, but that you keep track of which of them apply to which use cases.

[4] One exception might seem to be the "security" requirement. If the requirement specifies a particular sequence of events to the authorization, the flow of events may be affected, but in many cases there is simply a requirement that security must be guaranteed and it is up to the developers to ensure that the requirement is met. For these cases, making the requirement a "special requirement" of the use case is usually sufficient since including the security-related behavior merely gets in the way of understanding what the system is really supposed to do. These kind of requirements are often the most troublesome -- confusion over how to handle them often gets in the way of describing the real required behavior of the system.

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