Constantine & Lockwood, Ltd.

Structure and Style in Use Cases for User Interface Design

Larry L. Constantine
University of Technology, Sydney

Lucy A. D. Lockwood
Constantine & Lockwood, Ltd.

Abstract

Although widely employed in both object-oriented software engineering and user interface design, use cases are not well-defined. Little attention has been paid to the various styles for writing the narratives that define use cases and their consequences for user interface design and software usability. Common narrative styles are presented with examples and discussions of their relative advantages and disadvantages. Essential use cases, a variant employed within usage-centered design, are contrasted with conventional use cases and scenarios. For the most efficient support of user interface design and particularly for large, complex projects, a highly-structured form of use case has evolved. New narrative elements and relationships among use cases are introduced. These include means for expressing partial or flexible ordering of interaction, relationships with business rules, as well as a clarification of the often misunderstood concept of extension that recognizes two distinct forms: synchronous and asynchronous extensions.

Introduction

Since their introduction in support of object-oriented software engineering, use cases have enjoyed a seemingly explosive growth to become ubiquitous in both development methods and development practice. Part of this ubiquity can be attributed to their utility—use cases have proved to be versatile conceptual tools for many facets of design and development—but part may also be a consequence of a certain imprecision in definition. Most developers can say they are employing use cases because almost anything may be called a use case despite enormous variability in scope, detail, focus, format, structure, style, and content. Further muddying these already turbid waters, idiosyncratic terminology has been promulgated that obfuscates important distinctions, such as that between scenarios and use cases.

As an effective bridge between usability engineering and user interface design on the one hand and software design and development on the other, part of the promise that use cases offer is due precisely to their chameleon-like adaptability. For requirements engineering, use cases provide a concise medium for modeling user requirements; in the hands of user interface designers, use cases can become a powerful task model for understanding user needs and

Preprint of a chapter to appear in ***Object-Oriented User Interface Design,*** M. Van Harmelen, editor. The authors may be contacted at Constantine & Lockwood, Ltd., 58 Kathleen Circle, Rowley, MA 01969; tel: 1 (978) 948 5012; fax: 1 (978) 948 5036; email: larry@foruse.com | lucy@foruse.com
© 2000, L. L. Constantine

guiding user interface design; for software engineers, use cases guide the design of communicating objects to satisfy functional requirements. Success in all these endeavors rests on the realization that user interface design is not software design. Models originally developed to support the design of software components and their interactions are not automatically and necessarily well-suited for organizing user interface components and the interaction between users and these components.

Use cases undefined

One of the more remarkable aspects of use cases is that they have achieved such wide currency despite an almost complete lack of precise definition. Entire books have been devoted exclusively or primarily to use cases without even so much as offering a definition [Schneider and Winter, 1998; Texel and Williams, 1997]. Jacobson's original definition [Jacobson et al., 1992] is brief, broad, and barely descriptive:

> A use case is a specific way of using the system by using some part of the functionality. [A use case] constitutes a complete course of interaction that takes place between an actor and the system.

Owing in part to imprecise definition and in part to the confusion and conflation of the various possible uses and purpose of use cases, many use cases, including published ones, intermingle analysis and design, business rules and design objectives, internals and interface descriptions, with gratuitous asides thrown in to cover all bases. So deep is the confusion that even the most unconstrained mish-mash can be put forward as a use case.

Example 1.

> The guest makes a reservation with the hotel. The hotel will take as many reservations as it has rooms available. When a guest arrives, he or she is processed by the registration clerk. The clerk will check the details provided by the guest with those that are already recorded. Sometimes guests do not make a reservation before they arrive. Some guests want to stay in non-smoking rooms. [Roberts et al., 1998: 68]

What is the use case here, and what is its purpose? Is it to reserve a room or to get a room? Or is it to check in a guest? Who is the user and in what is role do they act? Is the user a guest or the clerk or the telephone operator who takes the reservation? What is the interface to be designed?

To their credit, the developers of UML (the hubristically monikered Unified Modeling Language) have collectively chimed in along common lines in an attempt to narrow the scope of definition somewhat. For example. a use case is:

> The specification of sequences of actions, including variant sequences and error sequences, that a system, subsystem, or class can perform by interacting with outside actors [Rumbaugh et al, 1999: 488].

In other works from the same group [Jacobson, Booch, and Rumbaugh, 1999: 41; Kruchten, 1999: 94], the definition has been qualified by the phrase "that yields an observable result of value to a particular actor."

Despite the addition of this slight nod to external significance, the current "official" definitions have actually moved away from Jacobson's original emphasis on use and have taken on what may legitimately be described as a more "system-centric" viewpoint: The focus is on what the system performs not what the user does or wants. In our opinion, this inside-out perspective, subtle though it may be, has actually contributed to problems in applying use cases to user interface design.

A somewhat more user-centric definition is offered by Fowler, in a popular introduction to UML [Fowler, 1997: 43]:

> A use case is a typical interaction between a user and a computer system … [that] captures some user-visible function … [and] achieves a discrete goal for the user.

For the most part, however, use cases have been defined not with reference to users but with reference to "actors"—external agents interacting with a system. For software design, actors may include other systems that must interact with the system being developed, but for purposes of user interface design, only the human users are, of course, relevant.

Most designers understand at some level that it is not so much users themselves but the roles that they play in relation to a system that must be taken into account in user interface design. The UML-sanctioned use of the term "actor" in this context is particularly unfortunate, because it leads to such awkward and ultimately nonsensical formulations as:

> An actor is a role that a user plays with respect to the system [Fowler, 1997: 46].

In all other areas of discourse, the actor is not the role, but is distinguished from it. The linguistic legerdemain of calling a role an actor is particularly confusing to users and clients, but it can also cast an insidious spell on the minds of designers who can, all too easily, unconsciously confuse the characteristics of an occupant of a role, the role incumbent, with aspects of the role itself. For this reason, we prefer to call the thing what it is and use the term role or user role to refer to a role played by human "actors," that is, users of a system. To avoid confusion and outright conflict with UML terminology, we distinguish system actors—other systems—from users in roles. A role thus constitutes a relationship between a user and a system and is defined by a set of characteristic needs, interests, expectations, behaviors and responsibilities [Wirfs-Brock, 1994].

Concrete and essential use cases

As most commonly used, use cases have described the actual interaction between external users (or system actors) and a system through a particular interface. Because they are expressed in concrete terms, such use cases are best referred to as concrete use cases. For example, consider the beginning of the "Returning Item" use case for a recycling machine [Jacobson et al., 1992: pp__]:

Example 2

> The course of events starts when the customer presses the 'start-button' on the customer panel. The panel's built-in sensors are thereby activated….

Use cases of this ilk are completely inappropriate for the design of user interfaces for the simple reason that they already assume some particular user interface and interaction design. To write use cases of this form, you must already have designed the user interface, at least in part. Why does the recycling customer press a button rather than a touch screen? Why does the customer have to do anything other than begin putting items in the recycler? The tacit assumptions built into such use cases can unnecessarily constrain the design and, in practice, often lead to inferior user interfaces with overly conventionalized and unnecessarily complex interaction.

It was precisely this dilemma of circular description that led us to devise a radically different kind of use case and to develop a systematic process for the creation and utilization of these use cases for user interface design. The secret for turning use cases into a truly effective tool for designing good user interfaces and enhancing usability is a shift in focus from interactions to intentions and from elaboration to simplification. Instead of modeling the interactions

between users and a user interface, the focus shifts to the intentions of users. Instead of elaborating use cases with specific details and alternative courses of interaction, the focus is on simplification, on simplified descriptions that capture the essence of a case of use.

We termed these models essential use cases [Constantine, 1994; 1995; Constantine and Lockwood, 1999] because they constitute essential models in the sense originally employed by McMenamin and Palmer [1984], that is, abstract, generalized, and technology-free descriptions of the essence of a problem.

We would now define an essential use case as:

> a single, discrete, complete, meaningful, and well-defined task of interest to an external user in some specific role or roles in relationship to a system, comprising the user intentions and system responsibilities in the course of accomplishing that task, described in abstract, technology-free, implementation-independent terms using the language of the application domain and of external users in role.

We were not alone in recognizing the need for such a teleocentric ("purpose-centered") approach to use case modeling and for a move toward abstraction in use case construction. Kaindl [1995] has proposed incorporating goals into use cases, as did Cockburn [1997], and Graham [1996] has argued the value of abstract use cases. A consensus in concept if not in details seems to be emerging: More recently, others have joined the chorus [Lee and Xue, 1999], and even the architects of the self-styled Unified Process have recognized the limitations of concrete use cases and the advantages of the essential form [Jacobson et al., 1999: 164].

It is our intention in this paper to help bridge the gap between software engineering and usability engineering by adding some clarity, precision, and depth to the discussion of use cases. Toward this end we will explore variations in narrative style and its significance for user interface design and will introduce the notion of structured use cases, use cases that are organized and described in a highly systematic form to enhance their utility both for user interface design and for integration with the rest of the development process.

Notation

We must make a somewhat apologetic explanation regarding the idiosyncratic notation employed in this paper. Although we recognize the hegemony of the UML, we have been somewhat tardy in reconciling our notation with that notation. While the UML pays close attention to defining the notational details of many of its models, almost nothing is specified regarding the notation for use cases, that is, for the specification that actually defines what a particular use case is and is about. This neglect of such a core issue has no doubt contributed to the confusion regarding what is and is not a proper use case.

Like all usability specialists and user interface designers, we are also keenly aware that the utility of information is profoundly affected by how it is presented. Notation is, in truth, the user interface of models. We have long argued that the visual form of notation can significantly increase or decrease the effectiveness of those who create and interpret the models [Constantine, 1994; Constantine and Henderson-Sellers, 1995a; 1995b; Page-Jones, Constantine, and Weiss, 1990]. Although some of the ideas in this regard have been incorporated into the UML, largely without acknowledgement, the bulk of the arguments and recommendations of those of us interested in the communicative function of notation seem to have fallen on deaf ears. Modern-day modelers are thus left with the choice between notational anarchy and a standard notation that is itself riddled with serious usability defects.

Two aspects of notation for use cases must be addressed. The first concerns the notation used within use cases, that is the style of representation employed in the narrative body defining the

use case. The second concerns how relationships among use cases are visualized in diagrams to convey the overall structure of the tasks being modeled through use cases. Each of these aspects will be addressed separately.

Usage-Centered Design

A usage-centered process

Essential use cases evolved within the context of an orderly process for user interface design, and, although it is not our intention to dwell on methodology, it is worthwhile saying something about that process. The process is called usage-centered design [Constantine and Lockwood, 1999] to highlight the fact that the center of attention is usage rather than users per se. Although our approach certainly partakes of a broadly user-centered design philosophy, in calling it usage-centered design we wanted to draw attention to those particular aspects of users that are most relevant to user interface design and to highlight the linkage to use cases as a model of tasks (or usage).

Usage-centered design is a model-driven process employing three abstract models: a user role model, a task model, and a content model. The user role model captures and organizes selected aspects of the relationship between particular users and the system being designed. The task model represents, in the form of essential use cases, those things that users in user roles are interested in accomplishing with the system. The content model represents the content and organization of the user interface apart from its appearance and behavior.

Each of these three models consists of two parts: a collection of descriptions and a map of the relationships among those descriptions. Thus the relationships among user roles are represented in overview by a user role map. The relationships among use cases are modeled in a use case map (corresponding to but not identical with the "Use Case Diagram" of UML). The content model represents the contents of the various contexts within which users can interact with the system, and the navigation map represents the interconnections among these interaction contexts. The content model with the navigation map constitutes what is sometimes called an abstract prototype [Constantine, 1999]. The "logical prototype" or "logical user-interface design" referred to in the Unified Process [Jacobson et al., 1999: 161ff] is essentially the same concept based on our work with Jacobson and his colleagues earlier in the 1990s [Ahlqvist, 1996].

Logically speaking, a final visual and interaction design, usually in the form of an annotated paper prototype, is based on an abstract prototype, that is, the content and navigation models. These, in turn, derive from the task model, which depends on the model of user roles. In practice, however, the role model, task model, and content models are built more or less concurrently, with the attention of the modelers moving freely among them depending on the course of analysis and problem-solving. The models themselves serve as holding places for the designer's fragmentary but evolving understanding, thereby helping to organize and systematize the process. This concurrent modeling approach makes for a more flexible and efficient process than traditional process models, which are more strictly sequential whether they are based on iteration or on a once-through "waterfall" cycle.

Usage-centered design is an "outside-in" approach in which the internal system design is devised to support the user interface, which supports external user needs. For this reason, essential use cases are considered the primary model, and concrete use cases, if needed for other purposes, are a derived model. In practice, use cases supporting system actors can be developed directly in concrete form and concurrently with the essential use cases supporting user roles.

Usage-centered design constitutes an adaptable agenda that can be practiced within almost any software development process [Constantine and Lockwood, 1999], including the increasingly influential Unified Process [Jacobson et al., 1999]. Unfortunately, the UP gets a few fundamentals backwards, such as making user interface design prototyping part of the requirements process. Nevertheless, with some improvements and refinements, the UP can serve as a context in which to practice usage-centered design. The details of an integrated process are beyond the scope of the present discussion, however, and merit full treatment in another paper.

Task modeling, scenarios, and use cases

Task modeling is the pivotal activity in usage-centered design. User roles are a useful construct, but the user role model is not so much of interest in itself. It serves primarily as a bridge to the identification and construction of use cases and secondarily as a holding place for information about users and their work environment as it affects user interface design. Content models are another bridge, connecting use cases to the user interface design. Both the role and content models facilitate the design process and promote systematic development of effective user interfaces, but they can be, under some circumstances, more or less dispensable. A task model, on the other hand, is indispensable.

Use cases and essential use cases are, of course, only one of many potential ways of modeling tasks, ranging from, on the one end of the spectrum, rigorous and highly structured approaches that are of greatest interest to researchers and academics, to, on the other end, informal movie-style storyboards and free-form scenarios. For user interface design, scenarios have been particular widely used.

Although some writers [Booch, 1994; Graham, 1996; Wirfs-Brock, 1993] have used the terms interchangeably, as most commonly employed in design [Carroll, 1995], scenarios are quite different from use cases. Scenarios are typically extended narratives forming a plausible vignette or story-line. They tend to be rich, realistic, concrete, and specific, often replete with gratuitous detail for verisimilitude.

For example, a scenario for “Ian Smith gets help with his HyperCam software installation through our new Web-based technical support system” might read something like this:

Example 3.

> It is 2 o’clock in the morning, and Ian cannot get his new HyperCam software to install properly. He points his browser to www.camerahype.com, gets the splash page, and waits for the corporate home page to appear. He scrolls down the page and clicks on the tech support link. On the provided form, he types his name, then gets his customer ID off the packing slip for the HyperCam and types it in. He clicks the Submit button. He scans the Tech Support home page and finally clicks on the .GIF showing a befuddled user with a packing crate. This takes him to the Installation Help page, where he begins filling out the Incident Report form. Dissatisfied with the suggestion supplied by the system after the form is submitted, he goes to the Contact Us page and sends an email message.

It is likely that the long-standing popularity of scenarios with traditionally trained HCI and user interface design professionals owes, in part, to this richness and realism. In contrast, use cases tend toward more stripped down and less interesting narratives, in which “variables” or class names replace more literal description and details are reduced, such as in this concrete use case for “getting installation help”:

Example 4.

> The use case begins when the customer goes to the Customer Log-On page. There, the customer types in his/her name and customer ID on the form and

> submits it. The system then displays the Tech Support home page with a list of Problem Categories. The customer clicks on Installation Help within the list, and the system supplies the Incident Report Form. The customer completes and submits the form, and the system presents a suggested resolution.

By contrast, a comparable essential use case for "getting help with specific problem" is even more spare:

Example 5.

user intentions	**system responsibilities**
identify self as customer	
	present help options
select help option	
	request description
describe problem	
	offer possible solutions

As these examples illustrate, scenarios, traditional (concrete) use cases, and essential (abstract) use cases represent successive levels of abstraction and generalization. A scenario comprises a combination of use cases as they are actually enacted by some user.[1] In this example, for instance, the scenario includes enactments of use cases for logging on, for getting help with a specific problem, and for sending email. Ignoring convention, the framers of the UML have declared that a scenario is a single thread through an instantiated use case; we prefer to honor accepted usage that is more common within the design and human-computer interaction communities, which would make a scenario a composition of enacted use cases, that is, a single thread through the instantiations of multiple use cases.

> A scenario is a composition of enacted (instantiated) use cases expressed as an extended narrative or sequence of events or states forming a plausible vignette or realistic story-line.

---

1. In object-oriented parlance, a use case is instantiated, but the term makes little or no sense to the average user. Human beings in user roles do not "instantiate" use cases, they "enact" or "carry out" or "perform" them.

Of the three task models, essential use cases are the most robust, especially in the face of changing technologies, largely because they model tasks in a form closest to the essential nature of a problem and do not intermingle design solutions with problem description. For example, the essential use case for "getting help with specific problem" need not be rewritten or altered in any way should the decision later be made to implement the tech support system as a menu-driven voice-response interface over the telephone. In contrast, neither the scenario in Example 3 nor the concrete use case in Example 4 would remain valid.

Use case decomposition

In usage-centered design, each essential use case represents incremental capability—a small piece of an aggregate set of tasks. Modeling with essential use cases favors a fairly fine-grained decomposition of tasks. The work to be supported by the system is partitioned into a collection of simple use cases interrelated by inclusion, specialization, and extension. Excessive decomposition of use cases has been denounced by some writers [e.g., Fowler, 1997], but decomposition in usage-centered design is not carried out for its own sake or to construct some sort of conceptually pure representation of a goal hierarchy.

Decomposition into relatively small, coherent units has several major advantages for usage-centered design. First, it leads to a smaller, simpler use case model in which each part of the overall model is comparatively simple and easy to understand in itself and in relation to a small number of other use cases. Second, the total model is simplified through reuse because common elements are factored out into separate use cases rather than being repeated in numerous places and variations.

When used as a guide to the design of the user interface and supporting internals, this form of decomposition promotes reuse of internal and external components and designs. Each piece of the problem can be solved once, and the solution can be recycled wherever the corresponding use case is referenced in the task model.

Our design experience also suggests a subtle but significant payoff in enhanced usability. A user interface that reflects a fine-grained, usage-centered task decomposition enables flexible re-composition by users into new combinations of tasks. In this way, designers can deliver powerful tools that support even unanticipated uses and usage patterns.

Use Case Narrative Style and User Interface Design

Scenarios, concrete use cases, and essential use cases are examples of varied styles of modeling and representation that have different relative advantages in different contexts. Usability inspections and usability testing usually require fairly detailed or specific scenarios that will exercise a variety of functions and expose a greater portion of the user interface to scrutiny. Non-technical end-users often are most comfortable with the greater realism and specificity of scenarios or concrete use cases. For software engineering and the design of internal software architecture, however, the more traditional concrete use cases have proven particularly effective. Essential use cases can be too abstract to guide many important programming and program design decisions, and scenarios informally intermix multiple functions, features, and threads of usage.

For user interface design, however, the abstraction of essential use cases is precisely the ticket, allowing the designer to model the essential structure of tasks without hidden and premature assumptions about user interface design details. We have long argued that abstraction encourages creative innovation, and the recent experiences of several teams using essential use cases have supported this argument with a string of new software patents.

Language and structure in models

Because language influences thought patterns, the style of writing, the format, the wording, and even the grammatical form employed in use case narratives can all influence the value of use cases for designing user interfaces. Of course, the same must be true for the design of software architecture as well, but our concern here is with usability and user interface design more than the other issues in object-oriented software engineering.

Given that really good user interface design is so difficult and real-world design problems are often so complex, the designer needs every bit of cognitive leverage and perceptual help attainable. Some styles of writing use cases facilitate good user interface design, while others are either indifferent or even interfere with it.

The more central and direct the role of use cases in the user interface design process, the more important becomes the issue of the form and style of representation. If, as in usage-centered design, the use case model directly drives and informs the user interface design, then narrative style and representation emerge as critically important.

Common narrative styles

Beyond the gross distinctions among scenarios, use cases, and essential use cases, we have found that the style in which use cases are written has a profound effect on their utility to designers and on the quality of the designs that result. In this section, we will illustrate some common styles in which use cases have been written.

The process or narrative body of a use case, what in UML is referred to as the flow of events, constitutes the definition of a given use case. Use case narratives have been written in widely varying styles that differ in a number of significant ways.

Continuous narrative

Many writers and modelers have favored a continuous, free-form narrative, a style first introduced by Jacobson and his collaborators. Here is a recent example taken from the Web:

Example 6.

> A cash withdrawal transaction is started from within a session when the customer chooses cash withdrawal from the menu of possible transaction types. The customer chooses a type of account to withdraw from (e.g., checking) from a menu of possible accounts, and then chooses a dollar amount from a menu of possible amounts. The system verifies that it has sufficient money on hand to satisfy the request. If not, it reports a failure to the session, which initiates the Failed Transaction Extension to report the problem. If there is sufficient cash, it sends the customer's card number, PIN, chosen account and amount to the bank, which either approves or disapproves the transaction. If the transaction is approved, the machine dispenses the correct amount of cash and issues a receipt. If the transaction is disapproved due to an incorrect PIN, the Incorrect PIN extension is executed. All other disapprovals are reported to the session, which initiates the Failed Transaction Extension. The bank is notified whether or not an approved transaction was completed in its entirety by the machine; if it is completed then the bank completes debiting the customer's account for the amount. [Bjork, 1998; used with permission]

The problems with this style of narrative are numerous. There is no clear separation between the user side of the interchange and the system side. The narrative intermixes internal and external requirements and jumps erratically between external and internal perspectives. Elements that are essential to the nature of the problem (e.g., "the machine dispenses the correct amount of cash") are co-mingled with implicit decisions about the design of the user interface (e.g., "customer… chooses a dollar amount from a menu of possible amounts"). The lack of structure forces the reader to trace through the entire text just to get an idea of the general nature of what is happening. Portions of the narrative that are important for the design of the user interface are buried among descriptions that are irrelevant.

Numbered sequence

Another common style is to write the narrative as a series of numbered steps. For example, consider another narrative for the same use case [Kruchten, 1999]:

Example 7.

Withdraw Money

1. The use case begins when the Client inserts an ATM card. The system reads and validates the information on the card.
2. System prompts for PIN. The Client enters PIN. The system validates the PIN.

3. System asks which operation the client wishes to perform. Client selects “Cash withdrawal.”
4. System requests amounts [sic]. Client enters amount.
5. System requests type. Client selects account type (checking, savings, credit).
6. The system communicates with the ATM network to validate account ID, PIN, and availability of the amount requested.
7. The system asks the client whether he or she wants a receipt. This step is performed only if there is paper left to print the receipt.
8. System asks the client to withdraw the card. Client withdraws card. (This is a security measure to ensure that Clients do not leave their cards in the machine.)
9. System dispenses the requested amount of cash.
10. System prints receipt.
11. The use case ends.

One advantage of this style is immediately apparent: The separation into distinct steps makes it easier to skim the use case for an overview and to see the general nature of the interaction. However, it suffers from many of the same problems as the continuous narrative style. Despite the segmentation into discrete steps, individual steps intermix system and user actions. The narrative exemplifies a systems-centric view: With one exception, each step begins with the system side of things.

Both examples 6 and 7 also illustrate an aspect of narrative style that is all too common in use cases. They are wordy and filled with verbiage that is little more than noise. Despite the fact that the beginning and end of a use case are invariably self-evident, many writers seem compelled to announce pedantically, “The use case begins when…” or to declare, “The use case ends.”

Partitioned narratives

Because they have little or no structure, both the continuous and the sequenced narrative styles require, for clarity, that the perspective or focus be declared repeatedly (the system does this, the user chooses that, the system completes something else), which contributes to their wordiness. Even so, the boundary between what is inside the system and what is outside—the user interface—is not readily apparent and can only be discerned piecemeal through careful perusal of the entire narrative.

The simple solution to this is to separate the user and the system side of the interaction completely. For use cases, this separation was originally suggested by Wirfs-Brock [1993], although others have effected similar divisions in other forms of task models. Wirfs-Brock divides the narrative of concrete use cases into two columns: the user action model and the system response model. In this style, the boundary representing the user interface is obvious, and it is immediately apparent, without intrusive repetition, which part of the narrative refers to the user and which to the system. For example, here is another variant of the cash withdrawal use case expressed as a partitioned narrative:

Example 8.

user action	**system response**
insert card in ATM	
	read card request PIN

enter PIN	
	verify PIN display option menu
select option	
	display account menu
select account	
	prompt for amount
enter amount	
	display amount
confirm amount	
	return card
take card	
	dispense cash if available

This format is so superior and more readable by all interested parties that it is hard to justify not using it in all cases. Of course, nothing prevents the use case writer from also numbering the actions and responses, which makes for even greater utility.

Pseudo-code

Some writers of use cases employ varying amounts of so-called structured English or pseudo-code in the use case narrative. For example, constructions like these are commonly encountered:

until customer_done repeat

if valid_user_code then do…end_do else do…end_do end_if

Although such expressions seem to offer precision and may be comfortable and familiar to software engineers, they are seldom very transparent to ordinary users, and the precision can actually obscure the real nature of the interaction.

In our experience, the more that use case narratives look like code, the more likely it is that they are just that. Programming is a necessary and noble activity, but it belongs as part of implementing a solution not in modeling the problem, which is yet another argument for essential use cases as the primary task model.

Interaction diagrams

Some designers substitute graphical models, such as the sequence diagrams and collaboration diagrams of UML, for the narrative text more commonly used to model the flow of events defining use cases. Although arguably well-suited to the original software engineering purposes for which they were conceived, interactions diagrams are a poor fit with the needs of user interface design. Instead of maintaining an external user perspective and a focus on the essential nature of tasks, they plunge the designer into considering messages passed among software objects. Like pseudo-code, they introduce internal design considerations prematurely, but the notation and constructs are even more obscure and alien to non-technical users and clients.

Pre- and post-conditions

One important exception to the rule against using programming constructs in use cases, particularly those intended for guiding user interface design, is the use of pre- and post-conditions, which have also become common in writing use case narratives. For example [Schneider and Winters, 1998]:

Example 9.

Place Order

Preconditions: A valid user has logged into the system.

Flow of events:

Basic Path

1. The use case starts when the customer selects Place Order
2. The customer enters his or her name and address.
3. If the customer enters only the zip code, the system will supply city and state.
4. The customer will enter product codes for the desired product.
5. The system will supply a product description and price for each item.
6. The system will keep a running total of items ordered as they are entered.
7. The customer will enter credit card information.
8. The customer will select Submit.
9. The system will verify the information, save the order as pending, and forward payment information to the accounting system.
10. When payment is confirmed, the order is marked Confirmed, an order ID is returned to the customer, and the use case ends.

Alternative paths

In step 9, if any information is incorrect, the system will prompt the customer to correct the information.

Postcondition: The order has been saved and marked confirmed.

Although we initially resisted this seeming intrusion of programming into task modeling, we found that pre- and post-conditions serve significant purposes in supporting usage-centered design. For one, ordinary users often are more comfortable when pre-conditions are made explicit. In the absence of the precondition in the above example, for instance, users and customers will often protest that the use case will not work or is incomplete without logging a valid user into the system. Indeed, many writers may be tempted to make such an action the first step of the use case, as was done in Examples 7 and 8. However, this shortcut has the disadvantage of making a discrete interaction a mandatory step in every use case dependent on it, which clearly can misrepresent the real task structure. (In most American ATMs, the user need not insert a card for each separate transaction, for example.) The pre-condition is left implicit in Example 6 (“A cash withdrawal transaction is started from within a session…”), which is significantly less clear than an explicitly declaration.

Pre-conditions also offer a non-procedural means for expressing necessary ordering among use cases. If “logging into system” is a use case modeled with an appropriate narrative, then a pre-condition expressed as “valid user logged in with ‘logging into system’” appropriately fixes the order of usage. These relationships among use cases are equally transparent and useful to users, user interface designers, and software developers alike. Contrary to the opinions of purists, this practice does not represent an intrusion of procedural modeling into object-orientation but merely expresses the intrinsic ordering of certain interrelated tasks. In fact, use cases as the pre- and post-conditions of other use cases provides a straightforward and logical means for modeling workflow, an aspect of task structure often neglected by use case

modelers. (An appropriate CASE tool would support tracing or highlighting woprkflow relationships.)

Abstraction

Use case narratives can be written at various levels of abstraction and to varying ends. In our own design work, we sometimes refer to a style we call "suitably vague," in which a certain amount of hand-waving and disregard for precision proves useful, especially in the early stages of task modeling. For example, in a numerically-controlled machine-tool application, an early form of one use case might read:

Example 10.

user intentions	**system responsibilities**
enter setup parameters	
	present setup configuration
confirm setup	
	perform tool setup

Not only does this degree of abstraction omit concrete details of the user interface design, it glosses over much of the content of the task in itself. There may be dozens of individual parameters that take on various foms and have various constraints. Presenting the setup configuration to the user may imply elaborate, unavoidable transformations guided by the user.

Nonetheless, such "suitably vague" modeling can be extremely useful in deferring a a premature digression into distracting details. In many cases, even obscuring essential details can pay off by implying the possibility of a generalized solution—for example, a common mechanism for entering setup parameters even though these are of varied form and format. Indeed, just such suitable vagueness led one team to a solution deemed by their legal advisors to be a patentable software innovation.

In essential use cases we avoid the extra words and verbal padding so common in most use cases for two simple and compelling reasons. First, phrases and constructions that add meaningless words merely decrease the signal-to-noise ratio and generally make it harder for analysts and designers to extract the real content that is relevant to user interface design. Second, substantive but unnecessary elaborations and redundant constructions, if actually translated into design features or elements, result in user interfaces that are unnecessarily complex.

A simple and representative instance is the use case for recycling (Example 2), in which the first user action represents an unnecessary step imposed by the narrative. Pressing a start button is not part of the problem as viewed from the user's perspective. Inserting an object to recycle is sufficient in itself to define the start of the use case. In practice, use case narratives reduced to essential form can guide the designer toward dramatic overall reductions in user actions, because every nonessential step in a use case model adds to the interaction design. One software tool for industrial automation, when redesigned through essential use cases, cut in half the total number of steps required to complete a representative mix of programming tasks. In fact, essential use cases have also proved an effective tool for workflow redesign and process re-engineering owing to their parsimony in expressing the inherent nature of tasks.

Task goals and user intentions

The narratives of essential use cases are cast in terms of user intentions and system responsibilities. Earlier in this chapter, we cited work on goal-orientation in use cases, but the distinction between goals and intentions is a subtle though important one for user interface

design. A goal is a desired end-state of a system, and as such it is correctly described in static terms as the state and features of objects. For example, in a hotel registration application, one goal of the hotel clerk might be expressed as “guest checked into acceptable room.” An intention, in contrast, is dynamic and represents direction or progress rather than an end state.

Goals are destinations, whereas intentions represent the journey, and it is the user’s journey—the interaction—that is most directly related to the issues of interface and interaction design. Goals, being static, place the focus on objects or nouns, while intentions, being dynamic, bring the actions and verbs to the foreground, implicitly admitting that interactions are alterable and divertible. Intentions may be satisfied short of reaching one particular final goal, and any number of intentions may support reaching a single goal.

Broadly speaking, both goals and intentions can be considered part of a broader “teleocentric” or “purpose-centered” perspective in use case modeling and user interface design. Beyond the immediate intentions of users, one can consider the contextual purposes of usage in a broader or larger perspective. In many applications, individual use cases are part of a larger framework of tasks, such as the work, profession, or job responsibilities of the user in a particular role that is being supported by the system. The larger purposes or functions of an essential use case are a part of the use case and need to be expressed within it.

In essential use case models, the complete hierarchy of goals and sub-goals, as well as that of tasks and subtasks, is expressed through interrelationships among use cases. These interrelationships are either embedded within the body of the use case narrative or are placed within specialized clauses or sections devoted to that purpose.

Structured Essential Use Cases

In any model-driven design process, the quality and character of the models shapes the quality and character of the results. For this reason, we have evolved a highly structured form of use cases based primarily on our own work and that of our clients in the practical application of usage-centered design but also incorporating ideas from numerous sources, including some of the stylistic innovations referred to in the previous section. The structure is intended to create an orderly and easily understood framework for organizing the various parts of the use case and to insure that all the relevant aspects of the use case have been defined—or at least considered.

Compared with the informality evident in most of the examples presented thus far, structured essential use cases form a more solid bridge between traditional requirements analysis and user interface design, and between design and implementation. As a tool for managing complexity, they also increase the effectiveness of essential use cases for truly massive problems. Informal and relatively unstructured forms of essential use cases are quite serviceable for relatively small and simple problems designed and implemented by small teams working in close communication. As the number of use cases and project participants increases, however, more systematic, elaborate, and highly structured forms of modeling and communication are needed.

There is no hard and fast rule for choosing between the informal and structured use cases. It depends as much on how the project is organized as upon the size of the problem. However, in general, modeling with informal use case narratives begins to break down with more than a few dozen use cases. Moreover, in any situation where analysts, designers, and developers work separately and communicate primarily through models, greater formality is also demanded.

The overall organization of a structured essential use case is illustrated schematically in Figure 1. The use case is divided into three principal parts: identification, relationships, and process.

The identification section uniquely identifies the use case and explains its purpose or function. The relationships section identifies other use cases related in some way to the use case. The process, the narrative body of the use case, defines its interaction process or dynamic aspects. Each of these will be explained in turn.

Identification	
ID	Name
Contextual Purpose	Supported Roles
Relationships	
Specializes	Extends
Resembles	Equivalents
Process	
Preconditions	
User Intentions	**System Responsibilities**
Asynchronous Extensions	Asynchronous Extensions
(steps)	(steps)
Post-conditions	

Figure 1 – Schematic framework for structured essential use cases.

Identity and Purpose

The Identifier is merely a sequence number or other unique and permanent identification code. While small problems may be modeled without resort to such codes, large projects will need them to help keep track of use cases and related models and documents.

The name of an essential use case should reflect its function or immediate purpose. An immediate purpose represents a single, discrete intention on the part of a user. Distinguishing intentions from goals leads to subtle differences in both description and definition of use cases.

We have found that present participles of transitive verbs—continuing actions taking a specific direct object—best capture the essence of user intentions. For example, getting cash from ATM or checking arriving guest into acceptable room. It is also important that the direct object be fully qualified by modifiers so as to distinguish it, as needed, from all other related or similar objects within the application. In general, especially as problem size goes up, more specificity in the objects is safer. Especially to be avoided are use case names that could be construed as referring to more than one operation of the same name on more than one distinct and unrelated object of the same name. For example, deleting block is unacceptable for naming a use case in an application involving both deleting program block temporarily, a reversible removal of a coding module, and the vastly different deleting execution-time block, the disabling of a run-time debugging break. The boundary between suitably vague and dangerously ambiguous depends on the specifics of the problem at hand.

One useful variation on ambiguous or non-specific naming is what we refer to as cluster cases, which are named collections of functionally related use cases. For example, downloading modules/blocks/programs to simulator/controller is shorthand for what might eventually

require six separate use cases or could potentially be covered by a single definition. Cluster cases may be based on a common operation on varying objects or on multiple operations on the same object, as in entering/modifying patient background information. Cluster cases are an especially useful shorthand notation in early modeling for applications with large numbers of use cases or for projects with extremely limited design time.

Having experimented with various orthographies for use case names, we now favor underlining and lowercase letters, with words separated by spaces. The underlining makes it easy to visually parse the use case name as a discrete whole distinct from the rest of the narrative in which it appears. Separating the words facilitates reordering and searching lists of use cases. Users also find this format to be “friendlier” and less “techie” than the strung-together, bi-capitalized style used in our book [Constantine and Lockwood, 1999] and earlier work. The underlining is also suggestive of a hyperlink, which is precisely how a use case name should function in on-line documentation or models.

The Contextual Purpose section contains the description of the larger goals or working context in which the use case is employed. For example, the Contextual Purpose of browsing empty rooms by category in a hotel reservation system might be: “In order to complete guest registration, a suitable room must be located and assigned to the guest.”

The Supported Roles section further qualifies the essential nature of the use case by listing the user roles that it is intended to support. For example, the use case browsing empty rooms by category might support the roles of Ordinary Desk Clerk, Desk Supervisor, and Self-Registering Guest.

Relationships

The Relationships section identifies other use cases to which the use cases is related. In usage-centered design, use cases may be interrelated in several ways:

- inclusion – one use case is included (by reference) within or used by another use case
- specialization – one use case is a specialized variant of another, more general case
- extension – one use case extends another by introducing alternative or exceptional processes

In addition, we find other relationships useful on occasion.

- similarity – one use case corresponds to or is similar to or resembles another in some unspecified ways
- equivalence – one use case is equivalent to another, that is, serves as an alias

Similarity, a relationship often noted early in use case modeling, provides a way to carry forward insight about relationships among use cases, even when the exact nature of the relationship is not yet clear. For example, reviewing savings account summary and reviewing checking account summary might be modeled as separate but similar use cases before it is known whether one process can cover both or if two different approaches will be needed. Equivalence flags those cases where a single definition can cover what are, from the user’s perspective, two or more different intentions. This construction makes it easier to validate the model with users and customers while also assuring that only one design will be developed. In the absence of such a relationship, the modeler would be forced to define the two use cases as trivial specializations of a totally artificial general use case.

Within the Relationship section, related use cases are listed in clauses labeled appropriately using the terms Specializes, Extends, Resembles, and Equivalents. Inclusions are omitted from

this section for the simple reason that these cases are, as the name implies, included in other parts of the use case narrative.

For example, the use case browsing empty rooms by category might incorporate the following Relationships:

Extends: checking arriving guest into acceptable room; reviewing room utilization

Specializes: browsing rooms

References within the Relationship section are reflected in the Use Case Map, which is discussed later in this chapter.

Process

The Process section of the use case begins with Preconditions and ends with Post-conditions. Both Pre- and Post-conditions can include explicit or implicit references to other use cases.

The narrative body of the use case is divided into User Intentions and System Responsibilities. Although we have primarily used the two-column format illustrated earlier in Examples 5 and 10, this arrangement can sometimes waste display space or present formatting difficulties, especially for use cases with complex steps. An alternative format uses indentation along with shading or distinct fonts to distinguish the System Responsibilities from User Intentions:

Example 11.

User Intentions

System Responsibilities

1. present list of standard test setups

2. select standard test setup

3. display selected test setup

4. optionally [do modifying test setup]

5. confirm test setup

6. run test as set up and report

7. optionally [print test results]

8. print and confirm

Although visually not quite as satisfying as the two-column format, this form is easier to create with word processors.

Elements of Style in Structured Essential Narratives

Within the process narrative describing user intentions and system responsibilities, the preferred primary language of expression is the natural language and vocabulary of the users and of the application domain. However, because essential use cases serve as the core model bridging between user requirements and the user interface design and between design and implementation, certain technical constructions and references are, of course, also required.

These technical elements include references to objects and other use cases as well as idiomatic constructions for expressing conditional interaction, partial ordering, and extensions.

Objects

Ideally, the use case model, whether in concrete or essential form, is developed in tandem with the object model or other data model. The steps of the use case narrative can make explicit or implicit references to objects, their methods, or their attributes. Thus, for example "test results" may be a defined term or object class, with "print" as one of its methods. To highlight a reference to an object or data element as a defined term, italics or quotation marks can be used. A use case as a whole may also correspond to a method of a class, as might be the case with the use case modifying test setup.

Included use cases

The process narrative may include explicit references to other use cases that are used in the course of or become part of the interaction. As illustrated in step 4 of Example 11, the name of the use case is preceded by the word "do," which calls out the included use case and improves scansion, especially when being read or reviewed with end users or clients.

Conditional interaction

Conditional interaction is presented with the word "optionally" followed by the optional actions in square brackets. Inclusions can be conditional, as in step 4 of Example 11, above. Conditional interaction is discussed further in connection with extensions, below.

Partial ordering

One of the shortcomings of most conventional ways of writing the narrative body of use cases is that they provide no explicit way to express a very common situation in task modeling. Most published narratives, including all of the examples thus far presented, are fully-ordered, describing strictly sequential interaction: first the user does this, then the system does that and that, and then the user does something else. In these narratives, some actions may be optional, but the ordering is not.

Although some analysts add procedural richness through iteration and conditional expressions, the issue of optional or flexible ordering is seldom modeled explicitly. This oversight is a major shortcoming, since many tasks can be performed in various orders, and allowing users to perform actions in various sequences is often a strong factor affecting usability.

In the past, many such cases were expressed in use case narratives as if they were strictly ordered even though it was implicitly understood that the order of interaction was actually flexible. In problems of modest size, one could trust that the final visual and interaction design would allow for the desired flexibility even though it was not expressed in the use case model. On large-scale projects, however, the chances are increased that important but implicit insights can be lost between analysis and design. The natural place to carry forward such information is as an intrinsic part of the use case narrative.

We have adopted the convention of using the phrase "in any order" followed by a list of actions or intentions, separated by semicolons and all enclosed in braces ("curly brackets"). For example:

Example 12.

in any order {specify part number; specify quantity ordered}

This practice encourages the user interface designer to model as strictly ordered sequences only those interactions where the order is actually fixed or required. The notation has proved easy and natural for users and non-technical people to understand without instruction. The

notation is designed to be distinctive enough that the words "in any order" can be omitted to form a shorthand that is convenient for specialists to use in rapid note taking or for communication among themselves.

Extensions

The notion of use case "extensions" has been a core concept in use case modeling since its introduction [Jacobson et al., 1992]. An extension is a distinct use case that embodies alternative, conditional, or exceptional courses of interaction that can alter or extend the main “flow of events” embodied in some base case. The base use case is said to be extended by the extension use case. The base case is the "normal" or expected interaction; the extension is an exception or alternative.

The primary advantage of extensions is that they make it possible to keep the narrative body of the base case simple and straightforward by separating out into distinct use cases the often numerous exceptional or unusual circumstances. Keeping the narrative for the base case clean is likely to lead to clean and simple operation of the user interface under normal circumstances. A second strong argument for extensions is that they encourage reuse—in models as well as in designs and implementation—since the extension case may modify or extend any number of other use cases.

Extensions pose a number of problems for methodologists and practitioners, however. One awkward aspect of the construct is the fact that the extension names the base case it extends, not the other way around as many people seem to expect. In other words, the extension is not visible in the base case, and diagrammatically, the arrow points from the extension to any use cases it extends. Although this convention seems backwards to many people, it is an advantage in one sense, since it allows extensions to be discovered or concocted after the fact without having to return to the base case and alter or rewrite the narrative body. Thus, initial attention is paid to the normal course of interaction and the special cases can be picked up later.

Although, even some of the leading thinkers in the areas of object-orientation and use case analysis have struggled over agreement on the exact semantics of extensions, we have always taken a purely pragmatic approach. Because they simplify task models and therefore can lead to simplified user interaction, usage-centered design employs extensions freely without worrying too much over semantic precision or theoretical rigor.

However, as the scale of design problems rises, with larger design teams and more and more use cases, the sort of "studied sloppiness" that can be beneficial for rapid design of modest problems begins to become a stumbling block. For this reason we have recently been rethinking the concept of extensions and how they are modeled in use cases and use case maps.

One of the problems is that the original concept of extension is actually a conflation covering two quite different kinds of exceptional or alternative courses of interaction. In the first variant, the extension use case can only occur at certain points in the course of the interaction, as in Example 11, where the user may or may not choose to modify the test setup. This is clearly not a part of the normal or mainline course of interaction, so it is appropriate to consider it as a distinct use case that extends running a standard test. However, the user can enact the extension use case, modifying test setup, only after a predefined test has been selected and before the test setup has been confirmed for the system to run it.

Synchronous extensions

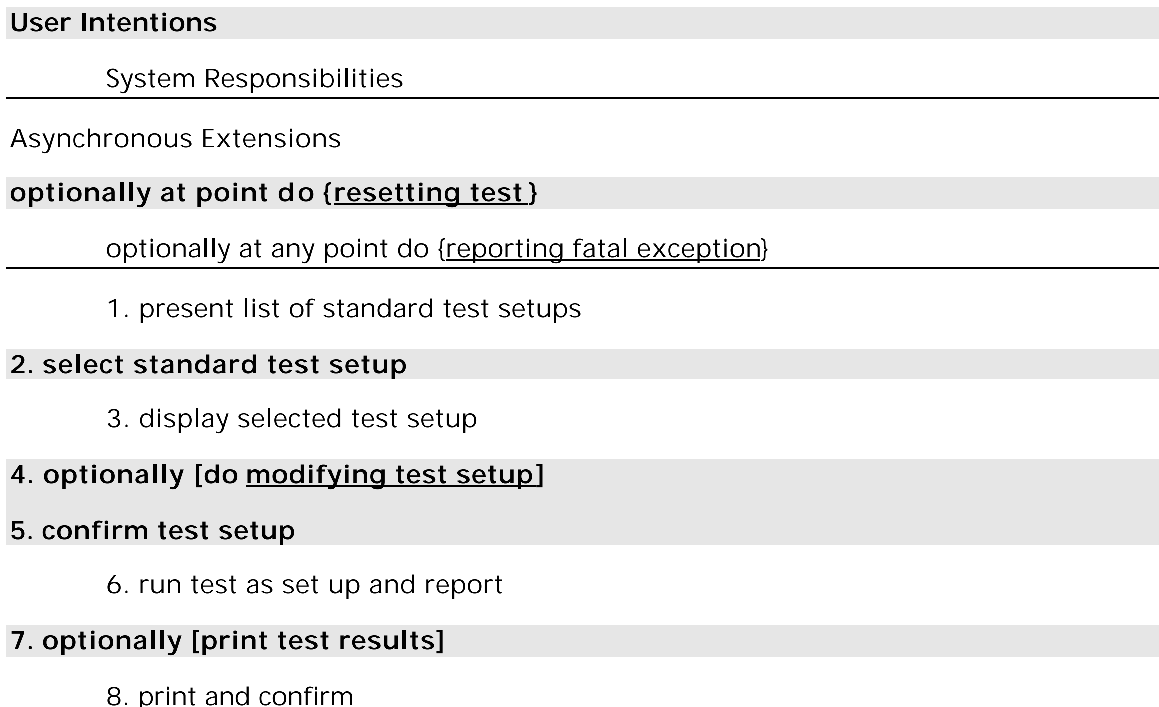
This sort of “extension” is termed "synchronous" because its occurrence is synchronized with the use case it extends. In fact, a synchronous extension is nothing more nor less than a conditional inclusion. It is a use case that will, under some circumstances, be included or used at a particular point in the course of the interaction. By definition it must be "visible" in the

narrative for the base case because this narrative must specify at what point in the narrative the extension can occur.

Asynchronous extensions

The other case covered by the original notion of extension refers to alternative or exceptional cases whose point of occurrence is unpredictable. Such asynchronous extensions operate like interrupts to the mainline interaction. For instance, in the use case of Example 11, the user might choose at any time to reset the system, canceling the test and restoring all original values. Similarly, the system could at any time assume responsibility for informing the user that an unrecoverable exception has occurred. Because such extensions apply to the interaction as a whole but can occur at any point, they are most reasonably separated out and presented at the top of the narrative body, as in this example:

Example 12.



User Intentions

System Responsibilities

Asynchronous Extensions

optionally at point do {resetting test}

optionally at any point do {reporting fatal exception}

1. present list of standard test setups

2. select standard test setup

3. display selected test setup

4. optionally [do modifying test setup]

5. confirm test setup

6. run test as set up and report

7. optionally [print test results]

8. print and confirm

We find that this arrangement puts the asynchronous extensions where users, managers, and non-technical people expect to find them. It is as if to say, "Before we start the details of our story, note that we could be interrupted at any time by certain occurrences." When showing use cases with to non-technical people, we would normally omit the heading “Asynchronous Extensions” from the section.

Use Case Maps

A use case map models the interrelationships among use cases, mapping the overall structure of the tasks to be supported by the system being designed. As noted earlier, problems with the UML Use Case Diagram have kept us from simply adopting its conventions wholesale. The large, complex models typically required for real-world applications can be particularly difficult to read and understand when expressed in UML. Integrating actors (user roles) with use case relationships in a single model leads to bewildering jumbles of lines for all but the most trivial problems of the sort found in books, articles, and tutorials. A suitable CASE tool would allow for selecting an actor or user role and seeing the associated use cases highlighted (or vice versa), but in paper or static diagrams, we would normally omit them.

Representing use cases

Following Jacobson's lead, we have always represented each individual use cases by an ellipse but have recently found that this convention, which seems adequate for academic or textbook problems where all the use case names are short and simple, breaks down for many larger and more complex real-world problems. In particular, the elliptical shape does not as readily accommodate long names as do other geometric shapes. Two alternatives that do not abandon convention altogether are worthy of serious consideration. One is to put the name outside the use case symbol, as Jacobson originally did. The other is to form a graphical hybrid that might be called an "elliptangle."

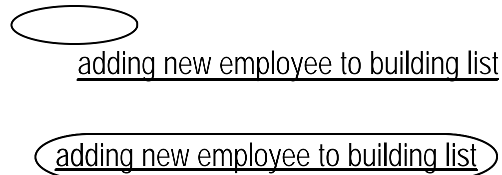


Figure 2 – Alternative representation of use cases in maps.

Cluster cases are portrayed as a graphical "stack," suggesting that the named case really covers a number of use cases.

Figure 3 – Representation of cluster cases in use case maps.

Representing relationships

Because in UML the type of relationship represented by a given line is not clear from its appearance, one must trace a line to its endpoints and find an adjacent label somewhere in the middle to know exactly how two use cases are related. The guillemets (« and ») surrounding relationship labels in UML are nothing more than graphical noise that makes models harder to parse visually.

As experience accumulates, especially with large, complex applications, we continue to revise usage-centered design notation to promote clarity and communication. For readability in complex models, we use different line styles to represent different relationships, employing labels for redundant communication. (See Figure 4.)

The distinction between synchronous and asynchronous extensions poses a notational challenge. The "extends" relationship is now reserved for asynchronous extensions. The fact that asynchronous extensions are referred to in the Asynchronous Extensions clause of the base case is considered a form of optional visibility, and the arrow still points from the extension to the extended base case.

Because synchronous extensions are merely conditional inclusions, they ought to be represented as some variation on the graphic for inclusion. We would prefer to model such relationships in a perfectly natural and expressive way by adding a decision diamond to the tail of the arrow normally used to show inclusion, which is precisely the way such relationships were represented in the structure chart notation in use around the world for more than 25 years [Yourdon and Constantine, 1975]. Unfortunately, the UML somewhat arrogantly usurps

the diamond—all but universally recognized as representing a decision or conditional operation—to represent aggregation, thus creating an arbitrary and non-intuitive notation made all the more obscure by the use of open diamonds and filled diamonds to represent (and, to many users, obfuscate) aggregation and composition of classes.

For conditional inclusion, other suggestions have been considered, including annotating the tail of the arrow with a question mark or with a cardinality, such as 0..1. We now favor a distinct line style with an informative label.

Specialization: specializes
Inclusion: uses (or includes)
Extension (asynchronous): extends
Inclusion (conditional): 0..N may use (or include)
Inclusion (as pre-condition): requires
Symmetrical relationships
Equivalence: equivalent
Similarity: resembles

Figure 4 – Relationships in use case maps.

Business Rules

In addition to functional requirements, almost all applications are defined in part by what are commonly known as "business rules." Business rules—or more simply, rules—are the policies, constraints, and conditions to be satisfied underlying business processes [Gottesdiener, 1999]. Rules define the operational relationships or restrictions governing information and how it is processed within the system. They are often referred to as business rules, even within technical applications not easily characterized in business terms, because they describe how "the business" or process is supposed to operate. For example, a rule might specify that new charges against accounts that are more than a certain period in arrears must be manually approved by a supervisor with specific authorization. Or a rule might require that passwords contain a mix of digits and letters and be updated at least every 90 days.

Rules can apply to or be associated with any part of a use case, a use case as a whole, or the relationship among use cases. The rules associated with use cases will have direct and important consequences for user interface design. A rule requiring regular update of passwords, for instance, mandates a means to inform the user when an update is needed and a means for changing the password accessible from the log-in context. To assure correct design and implementation and to facilitate tracing requirements, business rules should be connected with those parts of the analysis and design models to which they apply. On the other hand, business rules comprise a distinct collection of requirements that should not be scattered among various analysis and design models where they may become buried, lost, or

inadvertently duplicated. To resolve these conflicting needs, rules are compiled into a Business Rules model that identifies, categorizes, and lists them. Business rules can then be referred to by annotating use cases and use case maps with references to the rules by name and/or identifier.

For example, a business rule may require that a credit limit increase cannot be authorized on an account for which an activity summary has not been requested in the same session, even though, in principle, there is otherwise no necessary sequential dependency between these two functions. The use case reviewing account activity summary, therefore becomes a precondition of the use case authorizing credit limit increase. An annotation "(Rule: #27 Activity Review on Credit Increase) is associated with the relationship between the two use cases and would appear on the relationship line in the use case map and in the Preconditions clause in a form like this:

do [reviewing account activity summary] (Rule: #27 Activity Review on Credit Increase)

Recommendations

For user interface design, an appropriate style of expression in writing use case narratives can support a more efficient design process, facilitate communication within a project, and lead to higher quality designs. A consensus may be emerging that use case narratives that are both abstract and purpose-centered offer demonstrable advantages as task models for user interface design. We propose, additionally, that a more highly-structured and systematic form of use case narrative is needed, especially to support larger and more complex projects. Structured essential use cases provide a rich and precise mode of expression that not only meets the needs of user interface designers and of software engineers, but is also easily understood and validated by end users and clients.

Such use cases improve modeling and communication through a refined and precise organization that includes: (1) a clear division separating user intentions from system responsibilities, thus highlighting the system boundary; (2) distinction and separate presentation of asynchronous and synchronous extensions; (3) idioms for expressing partially ordered interaction; and (4) straightforward linking with related business rules. This structure strikes a balance that avoids both the informality and imprecision of continuous narratives on the one hand and, on the other, the inappropriate rigor of models, such as interaction diagrams, better suited for object-oriented design and programming than for user interface design.

References

Ahlqvist, S. (1996a) "Objectory for GUI Intensive Systems." Kista, Sweden: Objectory Software AB.

Ahlqvist, S. (1996b) "Objectory for GUI Intensive Systems: Extension." Kista, Sweden: Objectory Software AB.

Bjork, R. C. (1998) "Use Cases for Example ATM System." http://www.cs.gordonc.edu/local/courses/cs320/ATM_Example/UseCases.html

Booch, G. "Scenarios," (1994) *Report on Object Analysis and Design. 1* (3): 3-6.

Carroll J. M. (ed.) (1995) ***Scenario-Based Design.*** NY: Wiley, 1995.

Cockburn, A. (1997) "Structuring Use Cases with Goals," *Journal of Object-Oriented Programming*, Sep/Oct, 1997, pp. 35-40, and Nov/Dec, 1997, pp. 56-62.

Constantine, L. L., & Lockwood, L. A. D. (1999) ***Software for Use: A Practical Guide to the Models and Methods of Usage-Centered Design.*** Boston: Addison-Wesley, 1999.

Constantine, L. L. (1995) "Essential Modeling: Use Cases for User Interfaces," *ACM Interactions, 2* (2): 34-46, April 1995.

Constantine, L. L., and Hendersons-Sellers, B. (1995) "Notation Matters. Part 2: Applying the Principles," *Report on Object Analysis and Design, 2* (4):25-27, November-December 1995

Constantine, L. L., and Hendersons-Sellers, B. (19995) "Notation Matters. Part 1: Framing the Issues," *Report on Object Analysis and Design, 2* (3):25-29, September-October 1995.

Constantine, L. L. (1994) "Essentially Speaking," *Software Development, 2* (11): 95-96, November 1994. Reprinted in Constantine, L. L. ***Constantine on Peopleware.*** Englewood Cliffs: Prentice Hall, 1995.

Constantine, L. L. (1994) "Mirror, Mirror," *Software Development, 2* (3), March 1994. Reprinted in Constantine, L. L. ***Constantine on Peopleware.*** Englewood Cliffs: Prentice Hall, 1995.

Fowler, M. (1997) ***UML Distilled: Applying the Standard Object Modeling Language.*** Reading, MA: Addison-Wesley, 1997.

Gottesdiener, E. (1999) "Business Rules as Requirements." *Software Development, 7* (12), December.

Graham, I. (1996) "Task Scripts, Use Cases and Scenarios in Object-Oriented Analysis," *Object-Oriented Systems 3* (3): 123-142. 1996.

Jacobson, I., Booch, G., Rumbaugh, J. (1999) ***The Unified Software Development Process.*** Reading, MA: Addison-Wesley.

Jacobson, I., Christerson, M., Jonsson, P., and Övergaard, G. (1992) ***Object-Oriented Software Engineering: A Use Case Driven Approach.*** Reading, MA: Addison-Wesley.

Kruchten, P. (1999) ***The Rational Unified Process: An Introduction.*** Reading, MA: Addison-Wesley.

Lee, J., and Xue, N. (1999) "Analyzing User Requirements by Use Cases: A Goal-Driven Approach," *IEEE Software, 16* (4): 92-101, July/August 1999.

McMenamin, S. M., & Palmer, J. (1984) ***Essential Systems Analysis.*** Englewood, Cliffs, NJ: Prentice Hall, 1984.

Page-Jones, M., Constantine, L. L., and Weiss, S. (1990) "Modeling Object-Oriented Systems: A Uniform Object Notation." *Computer Language, 7* (10), October 1990.

Roberts, D., Berry, D., Isensee, S., and Mullaly, J. (1998) ***Designing for the User with OVID.*** New York: Macmillan.

Rumbaugh, J., Jacobson, I., and Booch, E. (1999) ***The Unified Modeling Language Reference Manual.*** Reading, MA: Addison-Wesley.

Schneider, G., and Winters, J. P. (1998) ***Applying Use Cases: A Practical Guide.*** Reading, MA: Addison-Wesley.

Texel, P. P., and Williams, C. B. (1997) ***Use Cases Combined with Booch OMT UML.*** Upper Saddle River, NJ: Prentice Hall.

Wirfs-Brock, R. (1993) "Designing Scenarios: Making the Case for a Use Case Framework," *Smalltalk Report,* November-December, 1993.

Wirfs-Brock, R. (1994) "The Art of Designing Meaningful Conversations," *Smalltalk Report,* February, 1994.

Yourdon, E., and Constantine, L. L. (1975) ***Structured Design (First Edition)***. New York: Yourdon Press.