OO design process: Use cases, an introduction
Designing the dynamic (runtime) behavior of a program

Allen Holub
Chief Technical Officer, NetReliance
January 2001

In previous articles, we've refined the problem statement and mocked-up our educational software. In this article we'll look at use-case analysis.

This article continues my series on the OO-design process. The first four parts are:

- Getting started
- Beginning to design software
- Refining the problem definition
- Verifying the analysis

Contents:

This month I move on to analyzing the dynamic (runtime) behavior of the program. I'll do that by introducing the notion of a *use case* and talking about how one of these beasts is organized. Next month, I'll put this organization into the framework of the Bank of Allen.

Introducing use cases

The "problem statement" looks pretty good at this point. I've probably forgotten something critical, but there's no point in sitting around hoping that these missing pieces will spring from my head, fully armed, like Athena from the head of Zeus. By moving on and looking at the same problem in a different way, these missing pieces will become evident. The next step, then, is to look at the problem again, but this time from the perspective of dynamic (i.e. runtime) behavior. In a sense, the problem statement is a *static* definition of the problem -- it describes the problem that the user has to solve, but that's it. Eventually, I'll formalize this view of the system into something called a *static model*.

The *dynamic* definition of the problem focuses, not on what has to be done, but on how you do it. That is, rather than looking at the problem itself, we'll look at the things the users have to do to solve the problem. The main tool we'll use to structure our thinking on that dynamic behavior is *use-case analysis*. Dynamic behavior can also be formalized into something called the *dynamic model*, but before we can do that, we have to figure out what the dynamic behavior *is*. We'll do this using use-case analysis.

First of all, a definition:

> A use case is a single task, performed by the end user of a system, that has some useful outcome.

The definition of "end user" and "outcome" can vary -- an end user might be an automated subsystem, for example, and the nature of the outcome can vary radically from use case to use case. Sometimes the outcome is a physical product like a report, sometimes is an organizational goal, like hiring an employee. But there's always an outcome that has a perceived (by the user) value.

You can think of a use case in terms of user intent. If a user walks up to the system with the intent of doing something, what is that "something?" Logging on is not a full-blown use case, for example, since nobody walks up to the system with the intent of doing nothing but logging on. You're logging on in order to accomplish some other larger end, to perform some useful task, and the use case is the definition of that larger task.

Presenting a use case

As is often the case in software, an organized semiformal presentation of the use cases helps you get organized. The formal presentation typically involves several pieces, listed in Table 1. (I'll explain what each section contains momentarily.) It's probably best to look at Table 1 as a checklist rather than a fill-in-the-blanks form. Not all sections are relevant in every use-case definition, and any of the sections can be combined (or omitted) in a real working document.

Moreover, though the categories in Table 1 must all be addressed, they don't need to be addressed in the order specified in the table. Often, it's best to start with the scenarios, develop them into a formal workflow definition, and then fill in the other details. The ordering in Table 1 is really for the convenience of the implementer, not the creator, of the use case.

We designers have several major goals in creating use cases:

1. Develop an increased understanding of the problem.
2. Communicate with the end users to make sure we're really solving their problems.
3. Organize the dynamic behavior of the design (and the resulting program) to reflect the actual business model in the user's mind. In other words, assure that the program actually does something useful for a real user.
4. Provide a road map and organizational framework for the actual development process. (More on this in a few months when we start talking about implementation.)

Table 1. The components of a formal use case presentation

Name
Description
Desired outcome
User goals
Participants/Roles
Dependencies
Preconditions
Scenarios
Workflow
Postconditions
Business rules
Requirements
Implementation notes

Use-case definition is a collaborative process. Many of these components are specified by the business-side (typically marketing) organization, and others are filled in or expanded by the technical-side organization. The user-interface design is also an important component of use-case analysis. A UI mock up (not a prototype) provides a lot of useful feedback about whether we've captured the use cases correctly. I'll come back to use-case-based UI design next month.

For now, let's expand on Table 1 and spell out the sections of the document in detail.

Name
All use cases should be named. Constantine recommends using a gerund followed by a direct object (for example: "withdrawing funds" or "examining the passbook"). This convention encourages the use-case name to succinctly identify the operation being performed and the object (or subsystem) that's affected by the operation. I think that this recommendation is fine as far as it goes, but there's no particular benefit to rigidly following a formula if you end up omitting important information as a consequence. For example, "customer withdrawing funds from checking account" might be a different use case than "bank manager withdrawing funds from customer's checking account." A simple "withdrawing funds" is not a sufficiently descriptive name.

Names should always be user-centric, not system-centric. For example, "making a deposit" (user-centric) versus "accepting a deposit (system-centric)." Name from the perspective of the user, not the system.

A name, regardless of format, is critical -- you must be able to identify the use case unambiguously to be able to talk about it effectively. I also typically assign some sort of administrative identity to every use case so that it can be referenced easily from other documents. I usually prefer combining a very general description with a hierarchical numbering scheme (e.g. "*Withdrawal: 1.1*").

Description
Describe what the use case is accomplishing. What will the user be doing while "withdrawing funds" or "examining a passbook," for example. Go into detail, but don't describe how the user might use a computer program.

For example, a bank customer might make a withdrawal by filling out a withdrawal slip and presenting the slip to the teller. The teller then takes the withdrawal slip to a bank officer for approval. The bank officer checks the account balance and issues an approval, etc. Note that nowhere in this discussion have I talked about computer programs, menus, dialog boxes, etc. These sorts of implementation details are *irrelevant* at this level. (Though, of course, you'll need well-defined implementation details before you can code. But we're not there yet).

I have a particular pet peeve about an analysis-level document that mentions "the system." It might be your problem to create a "system" of some sort, but it's certainly not your user's problem, and it's the user that's important here.

Desired outcome
By definition, a use case should have a useful outcome. Some sort of work that has value must be performed. Describe the outcome here. The outcome might be a report (in which case you should include an example of what the report will look like), an event or condition (an employee will now receive health benefits), or the like, but there must be a useful outcome.

User goals

What are the real goals of the user with respect to the use case? Note that goals are not the same thing as the use-case description. If the "Desired outcome" section describes *what* the user hopes to accomplish, the "Goals" section describes *why* the user is doing it.

Knowing the user goals can influence the direction of a use case in radical ways. To borrow an example from Alan Cooper, let's say that we are charged with creating a meeting-scheduler program. A tempting use case is "scheduling a meeting." Visions of calendars and appointment books immediately pop into your mind. But that way lies Microsoft Outlook -- a bloated, hard-to-use program that does virtually nothing useful for anybody (now tell us what you really think, Allen).

So how do we avoid the Outlookification of our system? By considering goals. What is everybody's goal with respect to a meeting? I guarantee that we all have the same goal: not to go at all. Barring that, our goal is to get out as quickly as possible, and to make the meeting as productive as possible. The only way to achieve that goal is to have an agenda. Therefore, the first step in the "scheduling a meeting" use case is the subcase "creating an agenda." (More about subcases in a moment.) The actual deciding on a date and time turns out to be secondary.

Participants/Roles

The participants in a use case are not physical users, they are the roles that a physical user might have with respect to a system. For example, if we're doing a time-sheet authorization system, two roles come to mind immediately: employees (who fill out time sheets) and managers (who authorize them). The fact that the same physical person might take on both roles at some juncture is irrelevant, there are two *logical* participants: the employee and the manager.

The UML characterizes these roles as "actors." However, I've deliberately avoided that nomenclature. To my mind, the physical user is an actor who takes on several roles with respect to the system. The actors are often outside the system, and are irrelevant. What's important is the roles that the actors take on. (I'm not alone in feeling that "actor" is a bad choice of terminology; see Resources.)

There is one important exception to the actors-are-external rule. Consider the issues surrounding access control: You are authenticating the actor to operate within the program in a given role. Consequently, the actor has a presence in the program, and a mapping of actors to roles is central to the way that the program works.

These categories are two components of something called a CRC (for Class/Responsibility/Collaborator) card, which are used in some OO-design methodologies such as Kent Beck's Extreme Programming. (CRC cards were developed by Beck and Ward Cunningham as a teaching aid; see Resources.)

Since this information is tied to participants rather than explicit use cases (that is, an actor in a particular role might participate in several use cases), it's sometimes useful to maintain the CRC cards as a separate document that's referenced from the individual use cases. Maintaining that list of responsibilities will be very important as we move to UML dynamic modeling.

CRC cards

For each role, we need to establish two critical pieces of information:

1. The *responsibilities* of actor when in this role. For example, bank tellers get deposit and withdrawal requests from customers, and get approvals for withdrawals from bank officers.
2. The actor's *collaborators* -- the roles with which communication is necessary. For example, a

Dependencies

Dependency relationships sometimes don't exist -- but that's unusual when a program is complex enough to have more than one

use case. Often the dependency relationships are not immediately apparent, but are discovered as the use-case model and associated user interface evolve.

bank teller collaborates with both the customer and the bank officer, but the officer never collaborates directly with the customer.

Typical dependency relationships include one or more of the following:

1. **Subset/Combines**
   A subset use case (which I call a `subcase`) typically appears when you start analyzing the top-level cases and discover that a complex task can be accomplished by performing several smaller, but standalone, tasks. Each of these standalone tasks is a subset of the main use case. Whether you use "subset" or "combines" is really just a matter of where you start. If you start with smaller use cases and realize that you can combine them together into a larger one, then use "combines." If you start with the larger case and decompose it, use "subset." For example, though identifying yourself (logging on) is typically not a use case in its own right (you wouldn't approach the system with the end goal of identifying yourself, and then shut off the machine and go home), it might be a subcase of other use cases. More to the point, the identification process might be the same in all use cases that required identification. It's useful to create an "identifying yourself" subcase and spec it out as if it were a standalone use case, then incorporate it by reference into the main use cases.
2. **Uses/Is-used-by** (includes)
   This one is very similar to a subcase relationship; don't waste a lot of time worrying about whether a "uses" or "subcase" relationship is better, since the two relationships are treated much the same way when using the use-case document. The main distinction between "subset" and "uses" is that a "uses" relationship applies when a use case is a subcase that is also a standalone use case. (It defines a standalone operation that has a useful outcome.) A subset-style relationship is sometimes used to distinguish a subcase that is used by only one other top-level use case, while a "uses" relationship might characterize a subcase that is used all over the place.
3. **Precedes/Follows**
   This establishes a workflow between use cases. For example, "registering a customer" must precede "specifying an order" or "browsing the catalog."
4. **Requires**
   Precedes/follows relationships indicate sequence, but not dependency. That is, "registering a customer" is required by the "buying items in shopping cart" use case, but it simply precedes the "browsing the catalog" use case -- it's not a requirement.
5. **Extends/Is-specialization-of**
   If use case B extends use case A (that is, adds subtasks, operations,etc.), then B is a specialization of A. (Typically the extra tasks are needed in order to satisfy some special requirement that doesn't occur in the normal use case). For example, "identifying a manager" might be a specialization of "identifying an employee" because the manager might have to be authenticated to a higher security standard than a normal employee. (This *specialization* relationship will be called *derivation* by programmer types, who would say that B "derives from" A to indicate that B is a specialization of A.)
6. **Resembles**
   Often, you'll notice that two use cases appear to be similar to each other, though there are minor differences in workflow. *Resembles* relationships indicate that you want to look closely at similar use cases, trying to find commonality that can be factored out into "subset" cases or

equivalents.

7. **Equivalent**
   Two use cases can appear to be different from the perspective of the user, but may end up being implemented identically (in the Bank of Allen example, a logical distinction between equivalent use cases). It's irrelevant to the user whether the underlying code is the same -- deposits and withdrawals are different logical operations. In any event, equivalent use cases have a way of diverging over the course of a design.

I've found that diagramming these dependency relationships using a UML static-model diagram is quite useful. (On the other hand, UML's official use-case notation is singularly worthless.) Constantine and Lockwood have proposed a notation for diagramming use-case relationships, but I prefer to use standard UML, introducing stereotypes when no existing notation convention can be pressed into service. I'll give you a concrete example later on (in the context of the Bank of Allen project).

Preconditions
What assumptions are you making about the state of the world when the use case runs? For example, customers must have an account with the bank before they can withdraw money. As a consequence, the "customer opening an account" use case must have been performed before the "customer withdrawing money" use case can be performed.

What conditions must exist before this use case can complete successfully? The conditions can be internal, or external. For example: The account balance must be greater than the withdrawal amount.

Scenarios
Scenarios are small narrative descriptions of someone working through the use case. Use a fly-on-the-wall approach: Describe what happens as if you're a fly on the wall observing the events transpire.

As is the case with the description, I try to keep the scenarios as abstract as possible (talking about how a bank, not a computer program that simulates a bank, is used). "Philip needs to make a withdrawal to buy groceries. He digs out his passbook from under the three-foot pile of dirty socks in the top drawer of his dresser, and finds that his balance is big enough to cover what he needs, and he heads off to the bank..."

Some programmer-types dismiss the scenarios as worthless fluff, but I've found them useful. The most important use is in clarifying my own thinking. Often, when working through a scenario, I'll discover use cases that I haven't thought of, or I'll discover workflow issues that were not apparent. For example, a couple of questions naturally arise in the previous make-a-withdrawal fragment. (What if he can't find the passbook?)

Consider a use case that might have several relevant scenarios. In a recent OO-design workshop that I conducted, we chose the class-registration system for Vanderbilt University as a sample project. There's only one high-level use case, here: "Registering for classes." Within this use case, several scenarios come to mind, however:

1. I sit down, sign up for all my classes, get into them, and I'm happy. This sort of scenario -- in which everything works without a hitch -- is called (at least by me) the "happy path" scenario.
2. I sign up for my classes, but one of them is full so I'm placed on a waiting list. Later on, a space becomes available in the class, I'm enrolled automatically, and am notified that I got into the class.

3. Same as (2) above, but the class is required to graduate -- I *must* get into it.
4. Etc.

In analyzing this set, I might decide that the second and third scenarios actually comprise a stand-alone use case that's distinct from the "happy-path" scenario.

Note that some scenarios are failure scenarios, but it's a mistake to think of the set of scenarios as containing a single "happy-path" scenario and a large number of failure scenarios. In the class-registration, example, one real failure scenario is that you have to get into a class to graduate, but you can't get into the class.

In fact, in most properly done use cases, there are no failure modes! This astonishing statement is not as shocking as it might, at first, appear. Most computer programs use error messages as a way to mask programmer laziness. The vast majority of "errors" could actually be handled just fine by the program itself. The error message is printed primarily because the programmers were too lazy to figure out how to handle the error, so they pushed the work onto the user. Take the class-is-full scenario. A lazy designer would say "Oh well, the class you want is full. That's just too bad. Nothing I can do about it" and classify this case as a failure scenario: A competent designer would work out a solution such as our waiting-list strategy. It's your job as a designer to solve the user's problems, not to push the problems back at the user. Alan Cooper goes so far as to say that well-written programs shouldn't print error messages at all. I'm not sure whether that goal is possible, but I'll bet you could eliminate 90% of the error messages from most programs, and end up with a more usable system as a consequence.

Also, for you programmer types, note that serious fatal errors -- the sorts of things that would cause exceptions to be thrown -- don't appear in the use cases at all. These conditions represent the failure modes of the program -- they are implementation related and have nothing to do with the problem domain, so they don't belong in a use-case analysis. Remember, we are defining the problem to solve from the perspective of the user, we are not designing a computer program (yet). That will come later, after we fully understand the problem. An exception-toss error condition is really an implementation requirement (see below) and should be listed in the "Requirements" or "Implementation notes" sections.

Don't take the foregoing discussion as an excuse not to think about failure conditions. Your goal is to think of every possible failure condition, and then make sure that every failure is handled gracefully every time it comes up. It can't hurt to make a working document on which you list all failure modes. You can then use that list as a checklist when you review the use cases to make sure that they're all covered.

As a final note, the sales staff will find the scenarios really useful in putting together presentations. It's always good to sell the product that you're actually building.

Workflow

Often the use-case description is sufficient to describe the flow of work through a simple use case ("do A, then do B, then do C"). Sometimes the workflow is so complex that it would be inappropriate to attempt to describe all of it in the Description section, and sometimes a diagram (like a UML workflow diagram, which I'll discuss eventually) is needed to unambiguously define the workflow. That sort of information goes here.

Postconditions

Frequently a use case changes the state of a system (i.e. the account balance is now lower) rather than generating a physical product. Also, some use cases need to be followed by others in order to complete an entire task. This sort of information goes in the Postconditions section. That is, a postcondition is something that you can check after the use case completes in order to determine the success or failure of the use case. An example postcondition: the new balance is the old balance, less the amount withdrawn.

Business rules
Business rules are policies that the business establishes that might effect the outcome of the use case. For example, "You may not withdraw more than $20 dollars within a seven-day period." It's not a good idea to clutter up the use-case description with these rules, but they have to be specified somehow to make the document accurate enough to be usable. I've found that it's best to specify these rules in a separate document and then reference the individual relevant rules here in the "Business rules" section.

Requirements
Requirements are typically customer-specified behavioral constraints on the implementation of a system. ("You must support 10,000 transactions per minute.") They are, again, best specified in a separate document and referenced here.

Note that some things that are called "requirements" actually aren't. UI design is often specified as a "requirement," but the real requirement is typically not how the program looks, but what it does. If the real users could care less whether the program uses kidney-shaped nested mauve buttons, then it's certainly not a "requirement" to use kidney-shaped nested mauve buttons, no matter how much you like them. (On the other hand, if the users do care, then it's indeed a requirement, no matter how silly it might seem to you. Live with it.)

I once had a client who was fond of the phrase "The users demand [some feature.]" My response was "Which users? Give me their names so that I can call them up and discuss it." Since no such users actually existed, it was easy to discount these demands. This is not to say that designers don't occasionally think of useful features, but unless a real user is *enthusiastic* about your innovations, then they aren't real. It's really easy to browbeat your user community to accept some gawdawful innovation that will do nothing but annoy them for the life of the program. Resist the temptation.

A good test for whether a requirement is valid is to reject any so-called requirement that specifies up front something that is a natural product of the design process (UI look and feel, program organization, etc.). Such "requirements" are just bad knee-jerk design. Expunge this junk from the requirements document. If you have a well-trained business-development team, these sorts of "requirements" won't be in the document to begin with.

Finally, note that it's impossible for an OO designer to work from a list of "features." Feature lists tend to be long, disorganized collections of poorly thought out ideas that some customer suggested to a salesperson off the top of their heads. The salesperson then translates that suggestion into "OH-MY-GAWD-WE-HAVE-TO-HAVE-THIS-NOW!!!," and away we go. At best, they suggest avenues of research, but by themselves they are of dubious value. What you're interested in is *why* the feature was suggested. How would this particular feature make the user's job easier? What is the user trying to accomplish? Do any of your other customers need to accomplish the same thing? Is there some better way to accomplish the goal? Most importantly, is the task that the user wants to do really a use case that didn't occur to us?

Implementation notes

Though it's our goal to keep the use-case document as grounded in the problem domain as possible, it's natural for implementation details to occur to you as you work through the scenarios and workflow. You want to capture this information when it pops into your head, so this section provides a place to jot down implementation notes. These notes aren't bolted in concrete. They aren't an implementation specification. Rather, they're details that will affect implementation and are relevant to the current use case. They will guide, but not control the implementation-level design.

Summing up

So that's the organization of a use case. Next month I'll show you how to apply the theory by developing a few use cases for the Bank of Allen project. Future articles will also show how to use the use cases for UI design, and how to build a formal dynamic model from use cases.

Resources

- Ivar Jacobson, Object-Oriented Software Engineering : A Use Case Driven Approach (Reading: Addison-Wesley, 1994; ISBN: 0201544350). This book is one of the first treatments of use cases. Though Jacobson is a pioneer in this field, I do have some arguments with him about things like use-case granularity. I've listed the book because it's seminal, but it's a bit abstract for my tastes.
- Larry Constantine and Lucy Lockwood, Software for Use (Reading: Addison-Wesley, 1999; ISBN: 0201924781). This book describes a use-case-based UI-design methodology called Usage-Centered Design. The approach is one of the best formal UI-design methodologies going, and the first quarter of the book is a good discussion of use-case gathering.
- Larry Constantine and Lucy Lockwood, Structure and Style in Use Cases for User Interface. This paper both summarizes and expands on the design approach discussed in Constantine and Lockwood's book. It's a good capsule introduction to the subject.
- Russell C. Bjork, An Example of Object-Oriented Design: An ATM Simulation. These are wonderful course notes for a class that Dr. Bjork is teaching at Gordon College. Though I have some criticism of some of the documents, Bjork does present a complete set of the design artifacts (including use cases) that arise from the OO-design process.
- Kent Beck and Ward Cunningham, A Laboratory For Teaching Object-Oriented Thinking. (From the OOPSLA'89 Conference Proceedings October 1-6, 1989, New Orleans, Louisiana and the special issue of SIGPLAN Notices Volume 24, Number 10, October 1989.) This paper describes the original use of CRC cards in a classroom situation.

About the author

Allen Holub is the Chief Technical Officer at NetReliance, a San-Francisco-based company that's building a secure global infrastructure for conducting trusted business transactions over the net.

He has worked in the computer industry since 1979, and is widely published in magazines (`Dr. Dobb's Journal`, `Programmers Journal`, `Byte`, `MSJ`, among others). He writes the "Java Toolbox" column for the online magazine JavaWorld, and also writes the "OO Design Process" column for the IBM developerWorks Components zone. He also moderates the ITworld Programming Theory & Practice discussion group.

Allen has eight books to his credit, the latest of which covers the traps and pitfalls of Java threading

(*Taming Java Threads*). He's been designing and building object-oriented software for longer than he cares to remember (in C++ and Java). He teaches OO design and Java for the University of California, Berkeley, Extension (since 1982). You can contact Allen via his Web site http://www.holub.com.

e-mail it!

What do you think of this article?

Killer! (5) Good stuff (4) So-so; not bad (3) Needs work (2) Lame! (1)

Comments?

Submit feedback