The Case For Using Use Cases

Stephen J. Mellor
Project Technology, Inc.

http://www.projtech.com
+1-(520) 544-2881
1-800-845-1489
steve@projtech.com

In which we describe how to apply use cases in requirements gathering and ensure requirements coverage; apply use cases for modeling (analysis and design); denote use cases in UML; model consequent behavior from use cases; and manage large numbers of use cases. In so doing, we shall make a case for when to use use cases, and perhaps more important, when not *to use use cases.*

1. BASICS OF USE CASES

Real-time and embedded systems respond to a large number of external events that cause the system to carry out some desired activity depending on the system's state. Each desired activity is a functional requirement. We may therefore come to understand system requirements by understanding the actors that generate external events, and the work that must be carried out in response—the use case.

Actors. The first step in use case analysis is to identify the actors that interact with the system and so require something from it. An actor is anything that interfaces and interacts with the system such as rôles played by people, machines, independent systems or subsystems, or sensors. Hence, in an elevator control system, actors include passengers, an external system reporting the speed of the elevator motor, and sensors that detect obstructions in the door.

Use Cases. A use case describes things actors want the system to do. In the elevator control system then, the passenger causes the system to request an elevator to a particular floor; the motor control box requires the elevator to open the door when the elevator comes to a halt; the door sensor detects an obstruction and so initiates door opening.

We represent the actors and use cases as shown in Figure 1. In the Unified Modeling Language (UML), we show the system as a whole as a rectangle. We show use cases as ovals inside the box, named according to the task to be carried out. The stick figures represent actors, named by the rôle that the actor plays. Note that the UML notation requires the use of a stick figure for the actor, even if the actor is not a person.

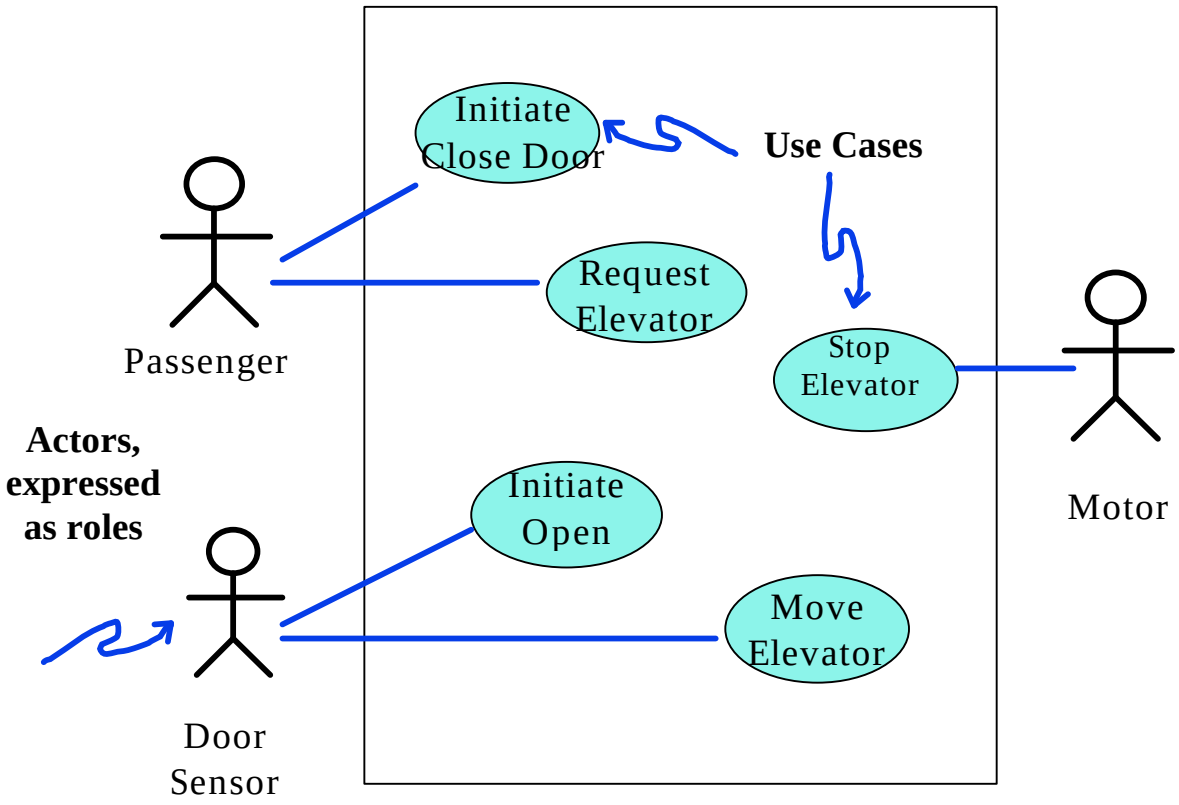


Figure 1: Some Use Cases for the Elevator Control System

2. How to Define Use Cases

There are many ways to write down the response for a use case,[1] and you may choose to invent your own. The approach selected here is based on pre- and post-conditions, which state what must be true for the use case to execute, and what must be true when the use case has completed. This has the advantage of both precision and concision, and in no way presupposes a particular design. Figure 2 shows an example.

Basic Structure.

To define the use case, use a combination of pre- and post-condition pairs, coupled with natural language text..

Pre- and post-conditions are grouped, so that we can say that if some pre-conditions are true then some group of post-conditions must also be true. There may be

[1] Jacobson[1] recommends describing a use case with a "flow description." However, in real-time systems, this implies including the sensor and actuator technology as a part of the requirements. We avoid this style because sensor and actuator technology frequently change both during development and deployment.

several groups, each of which describes what must be done in response to the same stimulus in different system states.

Request Elevator
Precondition: Elevator at floor with door closed
Postcondition: Door open initiated
Request satisfied

Precondition: No elevator at floor
Postcondition: Request queued for an arbitrary elevator

When a passenger requests an elevator, the system looks to see if an elevator is already there, with the door closed.
If so, that elevator's door is opened and the request is satisfied. If not, create a request for an elevator.

Figure 2: A description of the Use Case Request Elevator

We complete the use case with a natural language description of the desired behavior. This description must state clearly what the system has to do to achieve the post-condition. While the description must be detailed, it should not descend into other layers of the system. In the example above, for instance, it would be inappropriate to discuss sensor or actuator technology.

Pre-Conditions. A pre-condition denotes a state of the system that must be true if the use case is to be performed. The state may be compound and involve many instances, but it must not depend on the state of the actor. The stimulus that initiated the use case is implicit (there *had* to be one) and need not be repeated, but some find it helpful to do so.

This formulation ensures that the system is able to determine whether a pre-condition is true based on information available in the system, (i.e. it doesn't require "magic.")

Stimulus. Some stimulus initiates each use case. This stimulus may be a signal (a control value), the arrival of a data value (say, the number of the floor to go to), a data item reaching a value (position of the elevator reaching a floor), or the expiration of a timer (either absolute or relative).

The manner of the stimulus is not relevant to the essential requirements of the application, and so they should not appear as a part of the use case. This formulation effectively separates the interfacing technology from the problem.

For system engineering purposes, however, it is worthwhile to note how we intend to instrument the real world so that we can recognize the external event, and any

other proposed interfacing solutions. They should be noted as a part of a description of the external event, and not allowed to obscure the essential use case.

External Events. Whatever the stimulus is, it represents the need to execute some use case. We abstract the stimulus, in whatever physical form, as an external event. [2] An external event is the logical intent of the physical stimulus that initiates the use case. You may label the arrow from the actor to the use case with the name of the external event, though this is not a part of the formal definition of UML.

Event Recognizers. There may be computation involved in recognizing the event, for example, if the external event is the result of comparison of values. This is carried out by an event recognizer, which is not a part of the formal use case.

For system engineering purposes, however, it is worthwhile to note the requirement for an event recognizer and any additional communication required with actors. Similarly, if the only way to know whether a pre-condition is true is to examine the actor, then the system may need to track the state of an actor, or even interrogate the actor periodically. Again, these interfacing solutions are properly system-engineering issues, not requirements. They should be noted, as a part of a description of the external event, and not allowed to obscure the essential use case.

Timers. Use cases may be initiated by the expiration of a delay, or the arrival of some absolute time. Because 'time' is not a part of the system, it is possible to argue that it is an actor. However, one may reasonably argue that the timer mechanism is a part of the system, and that it's at best ridiculous to model the timer as a stick figure. Practically speaking, this is about as useful as arguing about how many angels can dance on the head of a pin. It really makes no difference. Invent a clock symbol; use a stick figure; don't show the timer. Whatever.

My own recommendation is to treat all use cases as requiring an external event. The external event requires recognition via an event recognizer, which may be trivial. The event recognizer may require signals or data from outside and inside the system, including the signal provided by a timer. We can then show the timer as a clock, hourglass, or the standard stick figure.

Establishing the Post-Condition. The post-condition represents what must be true when the use case has finished. Like the pre-condition, the post-condition must be achievable by the system without the help of actors. However, it is often the case that a particular potential use case requires an actor to do something. In this case, the best the system can do is initiate the desired behavior of the actor.

The formulation of a use case in terms of pre- and post-conditions avoids the pitfall of writing the specification of the use case in an overly procedural form.

3. BUILDING THE USE CASE MODELS

Now that we have established what a use case is, and how to write them down, we have the question of how we come up with use cases. Like any other creative activity, there is no absolute way to do this work. However, there are some techniques that we can use to kick-start the process.

Brainstorming Use Cases. Use cases are a requirements-gathering technique. Therefore, it helps to have users available to establish what the requirements are. In many embedded and real-time systems, we don't have "users" in the manner that business systems do. In the elevator control system example, a passenger may well know that he wants to go a particular floor, but that's not likely to be much help!

We therefore need to identify the experts who do have useful domain knowledge. These often include hardware experts, mathematical control experts, builders of the previous system, and so on. Invite identified users to a "brainstorming" session. Use a whiteboard or easel to record any results. Simply ask the users what the system needs to do. Don't criticize—let the ideas flow. When the session flags, identify actors, and ask what use cases they have, and so on.

The results of the use case brainstorming sessions provide a list of potential use cases of uncertain quality, so we next need to repartition, reorganize, and rethink. Three problems occur frequently, as discussed below.

Boundary-Crossing Use Cases. Preliminary use cases in real-time and embedded systems often require the actor to complete the activity. This is impossible: we may solicit some behavior, but we cannot guarantee that the actor will actually do the required work. For example, the incorrect use case "Passenger orders elevator to floor" requires the system to actuate the doors to cause them to close, to move the elevator to the floor, and to open the door again—all the while possibly stopping at intervening floors!

To address this seeming contradiction, break the use case down into several distinct use cases: Initiate Close Door; Initiate move to floor (when the door closes); Check for pending orders while at floor (when the elevator arrives at a floor, but still has outstanding orders); and so on. This approach has the added advantage of breaking a potentially large use case into several, potentially reusable parts. This approach also simplifies modeling of cancelled operations.

Vocabulary. Another common area of difficulty is the mixing of subject matters. The vocabulary of the use case should be completely within the subject matter at hand. Ensure that each use case uses an internally self-consistent vocabulary, so that no use case mixes subject matters. For example, "Passenger pushes button" mixes elevator vocabulary ("passenger") with sensor technology ("button").

We can formalize this notion of vocabulary with the notion of a domain, which is simply a subject matter.

Each domain has its own vocabulary, which must be internally self-consistent and complete. Verify that the use cases/actors match up with the abstraction level for the domain.

Inconsistent Abstraction Level. A related problem is inconsistency in abstraction levels. For example, a pre-condition requires the passenger to request an elevator, but it does not say how this request is made. It could be made by the passenger pressing a button; it could be done via graphical user interface, or it could even be effected by the Vulcan mind-meld. From the perspective of the elevator, the manner of the request is irrelevant. Therefore, to incorporate the sensor technology into the use case would mix abstraction levels.

For example, we earlier prohibited the use case "Passenger hits request button" because this did not identify what the system had to do, and it introduced a concept (buttons) that strictly speaking do not belong in the same subject matter as the elevator. In this case too, we must split the use case into several pieces, but it is not as simple as before because the use case involves two subject matters.

An effective approach to understanding abstraction is to view each domain as a system, and to view other domains as actors for that system. For example, we may treat the Process Input/Output domain as a system. "Passenger hits button" is still inappropriate because it still mixes domains, but we may modify this to "Button Pushed", or better "Toggle Device Pushed." This external event has a use case: "Register new state of toggle device" which can be described in the same manner as before.

Linking Use Cases Across Domains. One domain may make use of the use cases of another, so there is a link between "Passenger requests elevator" and "Toggle Device Pushed." Note that this is a derived requirement. The mapping between the domains at design time is another topic outside the scope of this paper.

Reorganizing the Use Cases. Now sort related use cases into domains. Use cases that are expressed in terms of button-pushing belong with a domain that recognizes such signals, in this system the process input/output domain. Use cases that refer to elevator concepts belong in the Elevator control domain, and so on.

4. BUILDING DOMAIN MODELS

Use cases provide an external view of the system, or more precisely, a domain. Whether we call the next step "analysis" or "design," in either case, we need next to construct a model of each domain—a domain model.

An object-oriented model of a domain comprises a set of classes and associations that declare the structural components of the domain. State models describe the dynamic behavior of each class.

Use Cases and Domain Models. The nouns in a use case description suggest classes, the verbs suggest associations, and the pre-conditions suggest states of those classes. However, this is little better than the discredited notion of underlining the nouns and verbs in a requirements document.

There are two problems here. First, the purpose of use cases is to present a view of requirements without becoming overly detailed and slipping into design. Use cases deliberately take an external, black box, view of the system. To build a good domain model then, we need to change our perspective from looking at the system as a black box from the outside, and instead look at the system from the inside as a white box.

The second problem is that use cases can hinder abstracting the requirements to the extent that the domain model does not use the user's vocabulary. At first, this may sound heretical. Surely, we *want* the domain model to match how the users think. Sometimes. Consider the following examples: an accounting system implemented as a spreadsheet; labels printed by setting the page length to 6 lines in a text-based formatter; the configuration of a chemical plant expressed in terms of territories and paths, rather than specific valves; a set-top box implemented as a Web browser; complex steps in a manufacturing plant expressed as macros comprising recipes and steps; and so on.

As a consequence, use cases provide the modeler with a start on understanding the problem and a few good classes, but they do not help *directly* with building the domain model.

Therefore, put the use cases aside when building the domain model. They will have served their purpose of gathering requirements. Now is the time to abstract.

Iterating the Models. Once we have a preliminary domain model, we can extend our understanding of the system by re-examining the use cases. Examine the domain classes to suggest likely use cases, and check the use cases to ensure you have the appropriate domain classes, too.

A reasonable process, which works in many situations, is described below. Every project is different, and the process described here is intended as a reference model—a way to get started, and a way to restart the process if it stalls. You should extend and modify this process as it works for you.

- For each actor that interacts with the domain, find the things the actor wants the domain to do.
- Write down the use case on the domain diagram.
- Specify the use case by describing the normal behavior first.
- Verify that the domain model has classes and states to support the use case.

- Specify the undesirable behavior for the use case.
- Repartition the use case to show different outcomes.
- Remodel the classes and states to support the variation.

Simplifying Use Cases: Managing Alternatives. A single use case may have several outcomes depending on state—particularly error conditions. Our example may be split up as follows. Notice how the state of the system is now inherent in the name of the use case.

Request Elevator with Elevator at Floor
Precondition: Elevator at floor
Postcondition: Door open initiated
Request satisfied

Request Elevator with no Elevator at Floor
Precondition: No elevator at floor
Postcondition: Request queued for an arbitrary elevator

Figure 3: A split use case, based on separating pre-conditions.

Alternatively, you may treat the alternative as "extending" the original use case. In this case, treat the error processing as a separate use case that "extends" the use case.[2]

Simplifying Use Cases: Re-Use of Responses. This example also suggests another situation where a use case may be required to be executed after another use case has completed. When an elevator is not available at the time of the request, the request is queued. When the elevator has come to a rest, without further pending orders, then the request should be dequeued, and the elevator should go to the requesting floor. That is, a use case may make "use" of another use case.

The diagrammatic form of a "used" use case shows the initiating use case with an arrow to the consequent use case, tagged with «uses». The « » indicate a stereotype (i.e. a special form or use of some UML element). The funny symbols are called guillemets ("ghee-yu-may" with a hard 'g').

The same strategy of simplifying by splitting use cases so responses can be re-used may be used anywhere in the system. Note, though, that excessive partitioning may encourage a functional decomposition of the system.

Completing the Model: Queries on Data. From whatever domain classes you have, go back to the use case to see if you have coverage.

[2] In business system use case modeling, it is common to model the interface as a part of the use case. Putting aside the inadvisability of this approach for the moment, this view lends itself to aborting the dialogue when an error condition is found. Because there may be several such errors, it is convenient to use the "extends" feature extensively. In real-time systems, though, it is often better to model faulty input in an event recognizer, and faulty system state as a pre-condition.

Check for coverage by ensuring that every use case has enough information, expressed as attributes of the classes, on the classes in the domain model.

Obversely, examine the domain classes and ask which use case creates and deletes each instance. Some instances will accumulate their attributes over time, each as a result of some use case. And some instances will exist before the system starts, and so will not have use cases associated with them.

Completing the Model: States. From whatever state models you have, go back to the use case to see if you have coverage.

Check for coverage by ensuring that every pre-condition and post-condition in each use case is expressed as a state or combination of states in the state models of the classes.

Obversely, examine the states in the state charts. Some use case caused transition into and out of the state. In some cases, the transition will have been caused directly by an external event, in others the transition will be the result of a signal event generated from another state chart, which was in turn triggered by an external event or other state chart. Some states self-generated events to cause transition out of the state.

Requirements Coverage. Once the reorganization and restatement of the use cases is complete, take them back to the users you identified earlier. Review them, and to see if any are missing. In particular, examine the pre-conditions to check that all possible states of the system have been covered.

Test Cases. The use cases form the basis for testing. Because each use case is a functional requirement, it is these that should be verified. Build your test plans by the domains in the system, starting at the lowest level, working up through the domains, preferring the process input/output, so that you can test the hardware, moving up finally into the application—the elevator control application.

5. THE MACRO PROCESS

Iterative and Incremental Software Process. This phrase rolls so nicely off the tongue that it's virtually a single word. Repetition and misuse have also made the phrase almost meaningless. Here are our definitions:

- Iteration is executing the *same* process on the *same* component.
- Incremental means executing the *same* process on *different* components

With these definitions in mind, we can see that an incremental process involves working different packages, while iteration means that you made a mistake, and so have to redo the work. This is not bad, per se, but it does say that less iteration means that you got it right earlier.

We recommend an incremental process, which will, unless you're perfect, also be iterative.

Domains provide a highest level system partitioning. For each domain, we follow a series of steps that leads to a complete, executable, system model. Domains that are large may be broken down into packages. Each package is a management convenience that is small enough to be understood as a unit, in every step, belongs only in a single domain.

We can represent the work on the project as a whole as a Project Matrix, in which the rows are the steps, and the columns are the packages. Each column is an increment.

Risk Analysis. Because there are several packages, we have to decide which ones to do first. Generally, we pick the riskiest first. If this is your first time using use cases, UML, or any new method, it may be advisable to treat the method itself as a risk, and tackle a simple package first to verify your understanding.

6. CONCLUSION

Use cases are a requirements gathering tool that facilitate communication with experts and users. Use cases describe what the system does from a black-box, external, point of view.

Use cases do not help directly with the abstraction required to build the domain models, which requires an internal, white-box view.

[1] *Object-Oriented Software Engineering: A Use Case Driven Approach.* Ivar Jacobson, Magnus Christerson, Patrik Jonsson, Gunnar Overgaard. Addison-Wesley

[2] *Structured Development for Real-Time Systems*, Paul T. Ward and Stephen J. Mellor, Prentice Hall