Modeling Alternative Courses in Detailed Use Cases

David Gelperin
LiveSpecs Software
dave@livespecs.com

Abstract

Real-life interactions between systems and users entail decisions and alternative courses of action. Within detailed use cases, there are four categories of alternative courses each having a distinct semantics. This paper describes the framework for categorizing alternative courses and provides modeling recommendations for each category. Finally, the paper examines the impact of alternative courses on use case conditions (i.e., constant, preconditions, and postconditions) and recommends a three-level strategy for specifying conditions.

1. Introduction

Real-life interactions between users and systems entail decisions and alternatives. Modeling detailed interactions requires the modeling of alternative courses of action. This paper describes a framework for categorizing these alternatives. Classification schemes not only promote consistent modeling, but aid in the systematic search for missing alternatives.

All alternative courses by the system are of interest. All user alternatives that trigger system alternatives are of interest. User alternatives that do not trigger system alternatives e.g., selection of a window, aisle, or center seat, are outside the scope of this paper, but are addressed in [6].

Modeling interactive usage with use cases is an important technique in requirements development. Use cases can and should be specified across a range of precision levels from short summaries that capture early concepts to comprehensive details that satisfy the needs of detailed system design, test design and system usage description. While decisions and alternatives may be omitted or imprecisely described in summary descriptions, they will be precisely specified in detailed use cases. The degree of precision used to describe decisions and alternatives, does not change the nature of those decisions and alternatives. Therefore, while this attempt to provide a framework for detailed alternatives may not be relevant to summary use cases, it is not affected by their existence.

While there is general agreement on many aspects of the specification of detailed use cases, some areas remain unclear. In particular, recommendations for specifying alternative courses differ among authors [1, 2, 3, 4, 5, 6, 8, 9, and 10]. The different recommendations are inconsistent and sometimes conflicting.

In aggregate, the following approaches are recommended:

(1) Selection constructs i.e., Ifs, in the basic course [2, 10],
(2) alternative course description(s) [all authors], and
(3) included use cases [all authors].

The major differences relate to the acceptability of using Selection constructs, the ways to use inclusion and extension use cases, and exactly how to model each kind of alternative course.

Part of the difficulty is that alternative courses can be put into at least four categories (described below), each having a distinct semantics. While it is well known that there are different kinds of alternative courses, this paper provides a comprehensive set of categories, with modeling recommendations for each. It also analyzes the impact of alternative courses on use case conditions i.e., constant, precondition, and postcondition, and provide a three-level strategy to repair the damage.

Although conditional repetition might be considered an additional kind of alternative course, it is outside the scope of this paper; however it is addressed in [6]. Note that it is possible to write alternative-free use cases, ignoring conditional repetition, by placing each distinct path in its own use case. However, this strategy leads to a very large collection of cases with small differences, so it is usually considered unwise.

The terms “alternative course” or “course”, where alternative is inferred, refer to an alternative path segment in a use case. Each alternative course contains one or more actions (steps) and is associated with a set of alternative conditions i.e., constant, pre, & post. Therefore, alternative actions appear in alternative courses.

Before looking at the different categories, consider the criteria for successfully modeling usage details. Detailed use cases precisely describe the interactions between a system and a set of actors. Successful models must be comprehensive, detailed, and accurate enough to satisfy the information needs of their neediest down-stream consumer, both present and future. Customers, testers, technical writers, and developers are all consumers of

requirements information. In addition, both reviewers and consumers must easily understand these models. This understanding depends on clarity and simplicity.

Readers should judge our recommendations for detailed usage modeling with these criteria. Compare the accuracy, clarity, and simplicity of these recommendations to your own current practice.

Readers are assumed to have a basic familiarity with use cases. Many of the references provide the necessary background. Cockburn [4] is recommended as a starting point.

The remainder of the paper is organized as follows: Section 2 contains parts of a sample use case used to illustrate approaches discussed in subsequent sections. Section 3 discusses insertion sites. Section 4 presents a framework for categorizing alternatives. Section 5 outlines the impact of alternative courses on case conditions and presents a three-level strategy for specifying conditions. Section 6 summarizes our recommendations.

2. A partial use case example

To ground the discussion, we provide the following partial example of using the reserve seat functionality of a Web-based Airline Reservation System, from [6]. In the rest of the paper, readers will find both small-scale examples and references to elements of this example. The referenced elements are color-coded for emphasis and ease of access.

Conditions (constant, pre, & post) in this example exist at three levels: the normal case conditions, plus **success conditions** **a**nd **alternative conditions**. The reason for the additional conditions will be explained in Section 6.

Case Name: Get [new] Seat on Reserved Flight

Risk Factors:

Frequency of occurrence: 0 to 2 times per reservation

Impact of failure: *likely case* – **low**, open seating is a workaround

worst case – **medium**, open seating in a large plane with many expensive seats is likely to anger important passengers

Case Conditions:

Constants:

None

Preconditions:

For reservation system, status is active

For passenger, system access status is signed on

Interactions:

Basic Course:

Passenger	**Web-based Airline Reservation System**
1. requests seat assignment	2. requests a reservation locator
3. provides a (corrected) reservation locator alternative	4. searches for reservation
Until (reservation located or all reservation locator strategies tried), actors repeat 3 to 4	
	5. offers seating alternatives, **unless** a. reservation not located or b. seat previously assigned or c. no seats are available or d. no seats are assignable
6. selects a seating alternative	7. assigns selected seat **unless** a. no seating alternative selected

	8. If (For reservation, seat previously assigned) returns previous seat to inventory *Post-conditions* -- For flight, previously assigned seat is available Endif
	9. confirms assignment SUCCESS EXIT

Basic Course Conditions

Constants:

For reservation system, status is active

For passenger, system access status is signed on

For passenger & flight, a reservation exists and can be located

Preconditions:

For flight, some seats are assignable

Passenger wants to get or change seat assignment

Post-conditions:

For flight, seating alternative was selected

For reservation, selected seat is assigned

Alternative Courses:

Exception Handlers (EH):

EH 1 - 5a (reservation not located) -- Handler omitted

EH 2 - 5b (seat previously assigned)

EH2 Constants:

For reservation system, status is active

For passenger, system access status is signed on

For passenger & flight, a reservation exists and can be located

For reservation, seat previously assigned

Passenger	**Web-based Airline Reservation System**
	1. offers to change seat assignment
2. wants a. to change seat assignment b. no change	3. If (Passenger wants to change seat assignment), CONTINUE Else, provides help and SUCCESS EXIT

Figure 1. Example of the reserved seat functionality in a web-based airline reservation system

3. Insertion sites

Every alternative course has one or more potential insertion sites i.e. points where the alternative may begin. These sites are located before, at, or after a process step. Sometimes, there is a single site and sometimes a range of potential sites. Sometimes the range includes all sites and sometimes only some of the sites.

Insertion sites can be identified implicitly by:

(1) placing a Selection or Case construct at the site e.g., basic step 8 above, or using an unless clause e.g., basic step 5 above,
(2) using italics for an abstract step name e.g., *pay for ticket* to signal a set of required selections, or
(3) placing a descriptive name at the site in square brackets to signal an optional action.

Sites can be identified explicitly by specifying the site locations along with the alternative course e.g., EH2 above.

Table 1: Categories for alternative courses

1	Exception Handlers
2	Required selections
3	Optional actions
4	User-invoked interrupts

4 Categories of alternatives

Remember that a precise use case contains one basic course, zero or more performed courses, and zero or more alternative courses. We recommend that the alternative courses be organized into subsections based on type.

We now describe the four types of alternatives.

1. **Exception Handlers** – Begin by considering exception handling (EH) alternatives. Handlers appear in most use cases to deal with abnormal conditions that may block the successful completion of a system process. An exception condition is checked at an exception site and if TRUE, the corresponding handler is invoked.

   We recommend a blended modeling strategy that places the body of the handler in a separate EH subsection and explicitly associates the exception condition with the course being protected by the handler. Consider using an "unless" clause as in basic step 5 above. Note that this step is explicitly associated with four exception conditions and therefore four handlers. This blended strategy makes analysis of step pre-conditions easier, while keeping process descriptions simple.
2. **Required Selections** – Next, consider a set of two or more mutually exclusive alternatives (RS), e.g., different ways to deliver an order, where one alternative must be done. It is possible to have "take no action" as one of the replacements, as long as it is not the most frequent alternative. If "do nothing" is the most frequent alternative, then it would be modeled by an optional action (next category).

   We recommend two approaches. For a few short alternatives, consider a Selection or Case construct, (as in handler step 3 above), placing the replacement steps inline. For many or longer alternatives, use a blended strategy that has the name of the abstract step in italics, e.g., *deliver the order*. This signals the existence of required selection in a separate RS subsection or in an included operation-level use case.

   If an exception condition can occur within a replacement alternative, avoid placing a process EXIT directly into the replacement's exception handler. Instead, associate an unless clause with the following action or use a "Continue, unless" construct. For example, there are many ways to pay for groceries. One replacement scenario is to realize that you have forgotten your wallet or purse and not pay. This can be modeled by following *pay for groceries* with a "Continue, unless not paid" and creating an associated exception handler.
3. **Optional Actions** – The third category is an optional action (OA). If the condition associated with the option is TRUE, then the action is included in the flow e.g., process a discount request. Shorter single-site options may appear directly in a Selection construct, as in basic step 8 above. We again recommend a blended strategy for longer single-site options, based on a suggestion by Constantine [5], in which descriptive names of options are placed at insertion sites in square brackets e.g. [process discount request]. This signals the option without burdening the process description. Often, the inclusion condition can be determined from the name. This bracketed name approach is only recommended for single-column descriptions.

   Some optional actions can occur at multiple sites in an inter-actor's process flow. For example, while

checking out groceries, a customer may realize that they do not have sufficient cash to pay for all of their selections and therefore ask that one or more items be removed i.e., ask for an adjustment action. This can happen at many points in the checkout process. A range of potential insertion sites should accompany each option. The range might include all sites.

Optional actions not directly specified at a site should be in a separate OA subsection or in an included operation-level use case.

4. **User-invoked interrupts** – We distinguish two types of interrupts, utilities and use cases.
   Utilities – On some software platforms, applications share a common set of utility procedures by convention e.g., save, cut, and help. We recommend that these interrupts be modeled with operation-level use cases that can be included.
   Application Use Cases – On some software platforms an application flow can be interrupted by use cases from any application. These application options (AO) are already modeled by use cases so that they can be included when modeling an interrupted flow.

5. A three-level strategy for specifying conditions

Case conditions are components of all standard use case formulations. In subsection 5.1, we explain why case conditions alone are too weak. In subsection 5.2, we recommend the addition of success conditions and alternative conditions and demonstrate their value by describing a process for deriving the conditions for any use case scenario based on these two additional conditions.

5.1 Case conditions are not enough

Pre, post, and constant conditions are used to explicitly define the guard conditions and effective consequences of a function or set of actions. Pre and post conditions have always been definitional elements of a use case. Constant conditions i.e., conditions that do not change from beginning to end, remove redundancy (same conditions appearing in both pre and post conditions) and strengthen the semantics of the conditions remaining in the pre and post conditions.

We use *scenario* to refer to a single complete path through a use case. Each scenario has its own set of pre, post, and constant conditions. A use case with alternative courses will have multiple scenarios.

Alternative courses weaken some or all of the case conditions. Alternatives weaken pre, post, and constant conditions for the use case because these conditions can only contain the conditions common to all scenarios through the case. Consider an exception handler that ends in a failure EXIT. Any scenario containing this handler course does not accomplish the objective of the use case. The post condition for this case can only contain what is common between a success scenario and all other scenarios, including this failure.

More alternatives mean more scenarios. More scenarios mean fewer conditions in common i.e., weaker case conditions. Weaker case conditions mean less information value in the case conditions.

5.2 Deriving scenario conditions

The following strategy enables pre, post, and constant conditions to be derived for each scenario in a use case. Begin by explicitly defining the conditions for the basic success scenario and for each alternative course in either an alternative subsection or included case. For examples, see the basic and exception handling subsections in section 2.

To recover scenario post-conditions from inline alternatives, such as basic step 8 above, we need a new construct that embeds post-conditions. For this we use the key word *Post-conditions* to indicate embedded post-conditions at the end of inline alternatives. If you don't use inline alternatives, you don't need this construct.

To develop a set of scenario conditions, use the following algorithm:

1. If the scenario ends at the end of the basic course, start with the basic success conditions otherwise start with the case conditions.
2. For each alternative course in the scenario, add the pre and constant conditions to the course preconditions and the alternative postconditions to the course postconditions.

Only use the next process step (3) for optional actions. In addition, use it for a group of exception handlers guarding one course, but only if all of the exception conditions in the group are FALSE. If an exception condition is TRUE and the corresponding handler is included in the scenario (just before the action), then step 3 does not apply to the remaining handlers in the group.

3. For each alternative course encountered, but not included in the scenario, add the negation of the alternative's pre and constant conditions *combined*. Combined means that if A represents a precondition and B represents an constant, the combined conditions would be (A and B) and the negation would be (not A or not B).
   a. If the non-included course is a single-site optional action or exception handler, add the results of negation to the scenario preconditions. For example, if a scenario encountered basic step 8, but its alternative course were not included, then (no seat previously assigned) is added to the scenario preconditions.

b. If the non-included course is a multiple site optional action, add the results of negation to the scenario constants.

4. Finish by (a) transferring conditions common to the pre and post conditions to the constant conditions, (b) deleting pre and post conditions that appear as constants, (c) deleting duplicate conditions, and (d) deleting the weaker of two conditions within the preconditions, postconditions and constants.

Using the algorithm to derive scenario conditions for successfully changing a seat assignment in section 2 starts with the success conditions and has the following conditions at the end of step 2:

Constants of changing seat scenario:
For reservation system, status is active
For passenger, system access status is signed on
For passenger & flight, a reservation exists and can be located
Preconditions of changing seat scenario:
For flight, some seats are assignable
Passenger wants to get or change seat assignment
For reservation system, status is active
For passenger, system access status is signed on
For passenger & flight, a reservation exists and can be located
For reservation, seat previously assigned
For reservation, seat previously assigned
Passenger wants to change seat assignment
Post-conditions of changing seat scenario:
For flight, seating alternative was selected
For reservation, selected seat is assigned
For flight, previously assigned seat is available

Step 3 would be skipped and the cleanup in step 4 would result in:

Constants of changing seat scenario:
For reservation system, status is active
For passenger, system access status is signed on
For passenger & flight, a reservation exists and can be located
Preconditions of changing seat scenario:
For flight, some seats are assignable
For reservation, seat previously assigned
Passenger wants to change seat assignment
Post-conditions of changing seat scenario:
For flight, seating alternative was selected
For reservation, selected seat is assigned
For flight, previously assigned seat is available

Consider what happens when using the algorithm for an "impossible" scenario that follows the same handler course for "a seat previously assigned", but skips the optional course in basic step 8. In step 3 of the algorithm, we would negate the precondition in basic step 8 and then cleanup would yield the pre-condition contradiction:

Constants of changing seat scenario:
For reservation system, status is active
For passenger, system access status is signed on
For passenger & flight, a reservation exists and can be located
Preconditions of changing seat scenario:
For flight, some seats are assignable
For reservation, seat previously assigned
For reservation, seat not previously assigned
Passenger wants to change seat assignment
Post-conditions of changing seat scenario:
For flight, seating alternative was selected
For reservation, selected seat is assigned

If the success and alternative conditions are consistent and logically complete, contradictions should always arise from impossible scenarios. Therefore, when a scenario that "can't happen" according to the customer does not yield a contradiction from the model's conditions, there is a bug in those conditions.

Since adding success conditions and alternative conditions amplifies the benefits of case conditions, we strongly recommend their use, even when deriving scenario conditions is not a goal.

6. Summary of Recommendations

The following tables summarize our modeling recommendations. Specific recommendations depend on the size or complexity of the alternative course and are discussed above.

Table 2. Modeling recommendations for alternative courses

Category		Basic Course	Alternative Course	Included Cases
1	Exception Handlers	unless clause	EH	
2	Required Selections	Select, *italics*	RS	operation level cases
3	Optional Actions	Select, [ ]	OA	operation level cases
4	User-invoked Interrupts			operation & goal level cases

In addition, we strongly recommend specifying success conditions and alternative conditions as well as case conditions.

8. Acknowledgements

Many thanks go to Ian Alexander, Keith Collyer, Don Firesmith, Jonathan Gelperin, and Oscar Coltell Simon for their many helpful comments.

9. References

[1] Steve Adolph and Paul Bramble *Patterns for Effective Use Cases* Addison-Wesley 2003

[2] Frank Armour and Granville Miller *Advanced Use Case Modeling* Addison-Wesley 2001

[3] Kurt Bittner and Ian Spence *Use Case Modeling* Addison-Wesley 2003

[4] Alistair Cockburn *Writing Effective Use Cases* Addison-Wesley 2001

[5] Larry Constantine and Lucy Lockwood "Structure and Style in Use Cases for User Interface Design" available at www.foruse.com

[6] David Gelperin "Precise Use Cases" available at www.livespecs.com

[7] Ivar Jacobson et al *Object-Oriented Software Engineering* Addison-Wesley 1992

[8] Daryl Kulak and Eamon Guiney *Use Cases: Requirements in Context* Addison-Wesley 2000

[9] Craig Larman *Applying UML and Patterns* Prentice Hall 2002

[10] Geri Schneider and Jason Winters *Applying Use Cases* Addison-Wesley 1998