Adding Bits of Precision to Usage Models

David Gelperin
LiveSpecs Software
dave@LiveSpecs.com

Abstract

For interactive systems, usage models are an important component of the requirements specification. In the spirit of agile methods and "just enough" written communication, well-understood user activities might be communicated by XP user stories and subsequent discussions. In cloudier areas, more details should be written to reduce the risk of misunderstanding. This paper describes nine techniques that you can use to increase the precision of your usage models. Each technique can be used alone or in combination to supplement your current practice. ***Readers are assumed to have a basic understanding of use cases.***

1. Introduction

For interactive systems, usage models are an important component of the requirements specification. Consider a defined set of stakeholders, their functional needs, and a set of system functions sufficient to satisfy those needs. Usage models can be used to demonstrate how the functions can be combined to satisfy the stakeholder needs and to demonstrate the operational completeness of the set of functions relative to the needs. Usage models enable users to see if the system functions fit their work styles. Such models also support user guide development, user interface design [5, 6], system and acceptance test design [4], the usage-based reading strategy for reviewing specifications [18], and the estimation of test and development effort [17].

In the spirit of agile methods and "just enough" written communication, well-understood user activities might be communicated by XP user stories and subsequent discussions. In cloudier areas, more details should be written to reduce the risk of misunderstanding. Even if a few sentences or paragraphs are sufficient to manage misunderstanding risk, more details can provide the benefits described above. The project team should decide how much usage information should be written. Sometimes this decision will be incremental as discussion reveals unanticipated complexity or misunderstanding.

Software development that produces code always gets to precision. So the question is not if precision, but when it first appears. Usage models with more details reduce ambiguity and therefore guesswork by software engineers. This does not mean that precision is necessary everywhere. The challenge is to provide "just enough" detail in "just the right" places to maximize effective communication and enable the success of those using the requirements information. Requirements users include customers, project managers,

© 2003 LiveSpecs Software

testers, technical writers, interface designers, software engineers, and specification reviewers.

In [3], Alistair Cockburn, introduces the notion of "levels of precision in functional requirements". In a subsequent wiki discussion [2], he writes about 1-bit through 6-bit precision corresponding to increasing information content. In his scheme for use cases, 1-bit precision names the goal, 2-bit adds the success course, 3-bit adds failure conditions, 4-bit adds failure courses, 5-bit adds object (data) descriptions, and 6-bit adds stakeholder models.

This paper describes nine additional candidates for details that you can add to increase the precision of your usage models. The first four candidates are additions to a use case specification while the other five candidates describe facts about the application domain. Each candidate can be used alone or in combination to supplement your current practices.

The remainder of the paper is organized as follows: Section 2 contains a sample use case that illustrates several of the addition candidates; Section 3 describes the four case condition candidates; Section 4 describes the five types of facts about the application domain; Section 5 contains conclusions.

In the following, readers are assumed to have a basic understanding of use cases [1].

2. Use case example

The following example describes the reserve a seat functionality of a Web-based Airline Reservation System [10] and illustrates some of the candidates for use case addition. We present the entire example without comment. Then we describe individual candidates and examine specific parts of the example.

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

© 2003 LiveSpecs Software

Interactions

Success 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

Success 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 description omitted

© 2003 LiveSpecs Software

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. Use case additions

We now describe the four candidate additions to a use case specification. They are (1) risk factors, (2) course conditions, (3) constant conditions, and (4) input alternatives.

3.1. Risk factors

To fully understand a user goal and the importance of supporting it in the context of a set of goals and support functions, one needs to know how often the user pursues the goal and the consequences of failing to achieve it. These factors should influence product interface and architectural design as well as verification and validation strategies.

Risk information, including both frequency of occurrence and failure impact (likely and worst case), should associated with each use case. This risk information should represent the post system reality, which may be quite different from the situation before the system exists. Note that all of this risk information reflects user activity in the application domain.

Failure likelihood has also been proposed for inclusion with use cases. While it measures an important form of risk, it is not risk associated with user activity, but risk associated with a particular system and its supporting technology. We therefore choose to package it with system design rather than use cases. Note that a use case reviewer that encounters failure likelihood values will require system design information to assess its accuracy.

© 2003 LiveSpecs Software

In the precise use case example above, risk information appears as:

> **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

We specify frequency in a range per event or per unit of time. Additional information such as average or mode values may also be included.

Failure impact is assessed as high, medium, or low unless techniques such as Failure Modes & Effects Analysis [7] or Hazard Analysis [16] yield more refined assessments. The rationale for each estimate is provided to make accuracy assessment easier.

If frequency or impact have multiple modes, then each mode and its conditions should be described. For example, tax-reporting functions may have extremely heavy use in November and February, medium use in December, March and April, and much lighter use in the other seven months. Averaging across modes does not provide useful information.

3.2. Course conditions

Most use case formulations suggest the inclusion of preconditions and post-conditions for each case. Unfortunately, case conditions alone do not usually contain much information, because they can contain only those conditions common to all paths through the case. Most cases contain paths representing accomplishment of the goals and others representing failure. There are few conditions in common between success and multiple forms of failure.

Pre and post conditions are useful ways to explicitly define the guard conditions and effective consequences of a sequence of actions. We propose that such conditions not only be associated with the case, but with each course of action in the case. This means associating conditions with the success course and with each alternative course.

Note that this does not mean explicitly associating conditions with each path through the case, because this array of paths is not explicitly modeled. Since a path is composed of a set of complete or partial courses, including course conditions enables the conditions for each path to be derived [13].

In the precise use case example above, the preconditions and post-conditions associated with the second exception handler would be:

© 2003 LiveSpecs Software

EH2 Preconditions

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

EH2 Post-conditions

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

3.3. Constant conditions

Constant conditions are those that are TRUE throughout a sequence of actions. We recommend that case and course conditions be specified using constant as well as pre and post conditions. The explicit specification of constant conditions removes the redundancy that must occur without them, i.e., an constant condition will occur in both the pre and post conditions.

In addition, factoring out constants strengthens the semantics of both pre and post conditions. Using constants, each precondition must have the potential to be FALSE at the end of some course of actions and each post-condition must have the potential to be FALSE as the start of some course of actions. Being clear about what may change and what may not has value in its own right.

In the use case example above, using constants for the second exception handler results in:

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

Using constants to specifying the course conditions for success results in:

Success 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

© 2003 LiveSpecs Software

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

3.4. Input alternatives

During use case interaction, user selections may be required between:

- available application choices, e.g. seat locations
- alternative ways to enter the same information, e.g. scan or key a credit card number
- alternative forms of the same information, e.g. see example below

Precision is increased by explicitly listing the user options in any of these situations. If the choices are common knowledge, then an explicit list is not necessary. Otherwise, explicit specification should be seriously considered.

Alternatives can be listed directly as in:

2. passenger wants
   a. to change seat assignment
   b. no change

Alternatives can also be named and then listed in an application domain glossary. For example:

3. passenger provides a (corrected) *reservation locator* alternative

with glossary entries:

reservation locator

a. reservation confirmation number
b. **flight id** and **passenger id**

flight id

a. flight date, departure airport, and flight number
b. flight date, departure airport, and destination city

passenger id

a. frequent flyer number
b. passenger name

Note that there are 5 valid forms of reservation locator.

© 2003 LiveSpecs Software

The selection of directly listed or named alternatives depends on complexity.

4. Application domain facts

We now describe the other five candidates. Each is a type of fact about the application domain. These candidates are (1) derived values, (2) derived conditions, (3) function limits, (4) domain constants, and (5) condition dependencies. As with named alternatives, these candidates might also be placed in an application domain glossary. Some readers will recognize these candidates as business rules [14, 15].

4.1. Derived values

The values of some attributes may be derived from the values of other attributes. Derived values explicitly show the calculation, which may include any mathematical function or operation.

For example in a library management system, the total number of copies might be specified as follows:

> Total # of copies (i.e., current inventory)
> = (# of available copies + # of open reserve copies
> + # of closed reserve copies + # of onloan copies
> + # of inrepair copies)

4.2. Derived conditions

Derived conditions are analogous to derived values. The truth-value of the condition is derived from an expression containing other underlying conditions. Examples include (applicant is eligible) and (order is valid). Derived conditions increase readability without sacrificing precision.

A derived condition is shorthand for a logical expression containing underlying conditions joined by logical *AND*s and *OR*s. Understanding the full meaning of the derived condition requires understanding the underlying conditions and their logical relationships. (order is valid) can be defined by "*AND*"ed sets of valid underlying conditions that are "*OR*"ed together, i.e. conjunctive normal form.

(platform has failed) could be defined as ((power has failed) OR (hardware has failed) OR (system software has failed) OR (communication has failed)). Any of the underlying conditions might be further decomposed.

4.3. Function Limits

There are often limiting or boundary values that control the behavior of a function. These limiting values should be associated with explicit attributes of individual transaction

© 2003 LiveSpecs Software

objects or aggregates of these objects. Examples include aggregate reservations limit for an aggregate of flight reservations, purchase limit for a single purchase, loan limit for a single loan, and loan type aggregate limit for all loans of a particular type. We recommend the explicit specification of function limits because too often such rules only exist implicitly in the processing code.

As with other attributes, these values might be bounded or derived. For example, in many airline reservations systems, there is a derived value rule like the following:

aggregate reservations limit:
for each flight, the number of active reservations must not be greater than
(100 + oversell percentage) x (total number of usable seats).

This limit constrains the reservation booking function.

4.4. Domain Constants

In an application domain, each class has attributes and definitions for valid values of these attributes. In addition, there may be additional (business) rules that define constant relationships between attribute values within a class or across classes. These rules may be conditional, i.e. TRUE under some conditions, or unconditional, i.e. always TRUE.

Domain constants are unconditional rules defining constant relationships between attribute values across classes. For example:

For every book:
no patron borrows more than one copy
only librarians can reserve, fix, or remove copies
For every flight:
no available seat is assigned
no assigned seat is available
no seat is assigned to more than one passenger

Although domain constants could appear among the constant conditions in every use case in the application domain, we recommend that they be specified with class definitions or in an application domain glossary.

4.5. Condition Dependencies

Sometimes there are dependencies between conditions that should be explicitly documented. For example, in the success course of the use case above, we find:

© 2003 LiveSpecs Software

5. *Web-based airline reservation system* 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

There is a dependency between the last two conditions.

If "no seats are available" then "no seats are assignable".

This means that if the third condition is TRUE, then the fourth condition must also be TRUE. If seats are available however, seats may or may not be assignable.

We recommend that inter-condition dependencies be explicitly specified because, as with domain constants, it increases their visibility and supports their effective review.

5. Conclusions

One or more of the nine detailing candidates may be used to help avoid recurrent or anticipated misunderstandings or to increase the benefits of usage modeling.

6. References

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

[2] Cockburn, Alistair "Comments on User Story and Use Case Comparison" wiki page Available at http://c2.com/cgi/wiki?UserStoryAndUseCaseComparison)

[3] Cockburn, Alistair "PARTS: Precision, Accuracy, Relevance, Tolerance, Scale in Object Design" Available at http://members.aol.com/acockburn/papers/precisio.htm

[4] Collard, Ross "Developing Test Cases from Use Cases**"** Software Testing & Quality Engineering July/August 1999 pp. 30-37

[5] Constantine, Larry and Lockwood, Lucy **"**Structure and Style in Use Cases for User Interface Design**"** Available at www.foruse.com/Resources.htm#style

[6] Constantine, Larry and Lockwood, Lucy **Software for Use** ACM Press Addison Wesley 1999

[7] Failure Mode and Effect Analysis (FMEA) Information Centre (www.fmeainfocentre.com)

[8] Gelperin, David "A Q&A Intro to Precise Usage Modeling" Available for download at www.LiveSpecs.com

[9] Gelperin, David "Just Enough Precision" Available for download at www.LiveSpecs.com

[10] Gelperin, David "Precise Use Cases" Available for download at www.LiveSpecs.com

© 2003 LiveSpecs Software

[11] Gelperin, David "Precise Use Case Examples" Available for download at www.LiveSpecs.com

[12] Gelperin, David "Precise Usage Model of a Library Management System" Available for download at www.LiveSpecs.com

[13] Gelperin, David "Modeling Alternative Courses in Detailed Use Cases" Available for download at www.LiveSpecs.com

[14] Gottesdiener, Ellen **Business Rules show Power, Promise** Applications Development Trends March 1997 [www.adtmag.com/pub/mar97/softeng.htm]

[15] Guide **Business Rules Project – Final Report** October 1997 Available at www.essentialstrategies.com/publications/businessrules/index.htm

[16] Leveson, Nancy G. and Peter R. Harvey, "Analyzing Software Safety," *IEEE Transactions on Software Engineering*, Vol. SE-9, No. 5, September 1983.

[17] Schneider, Geri and Winters, Jason P. **Applying Use Cases** Addison Wesley 1998

[18] Thelin, Thomas, Runeson, Per, and Regnell, Bjorn "Usage-based reading – an experiment to guide reviewers with use cases" Information and Software Tech. 43 (2001) pp. 925-938

© 2003 LiveSpecs Software