Use Cases that Work

Using Event Modeling to infuse rigor and discipline into Use Case Analysis

David A. Ruble
Principal Senior Partner

Olympic Consulting Group
Software Architecture and Development

P.O. Box 4008 – Federal Way, WA 98063
(253) 946-2690
www.ocgworld.com

Use Cases are widely used as one of the primary analysis models in system development, however, many analysts struggle with how to create a set of use cases, and how best to document each use case.

In this paper, OCG's David Ruble shows how Event Modeling can be used as a proven method and best practice for discovering and documenting use cases.

Use Cases that Work

Using Event Modeling to infuse rigor and discipline into Use Case Analysis

Over the years, OCG has been involved with many clients who are using use cases as one of the primary analysis models. Our experience with use case modeling has identified two major issues. The first is the difficulty in determining what constitutes a use case. The second is how best to document the details of a use case once you've got your hands on one.

What is a use case anyway?

Consider the original definition of a use cases:

> *"A use case is a sequence of transactions in a system whose task is to yield a measurable value to an individual actor of the system."*
>
> *– Jacobson et al., 1995*

Like the classic Rorschach test, one can stare into the inkblot of this definition and see just about anything. The use case definition is very general because it is trying to cover virtually all types of interactions with all types of systems. The first problem we encounter as people try to define use cases is that there are few guiding principles as to what a use case is or is not. For the development of new business systems, this very general definition is simply not sufficient. At one client, a group of programmers was sent to a half-day seminar on uses cases, in which they were taught to trawl through requirements documents to look for verb-object combinations that have been uttered by users. "These will be your use cases," beamed the fresh-faced young instructor. With the incandescent glow of enlightenment, the team repaired to their cubicles and created a list of use cases that included:

Destroy opposition,
Interoperate with other systems, and
Be user-friendly[1]

At other clients, we have encountered projects with similarly vague verb-object combos, including some of our favorite offenders:

Make request, and
Get answers

Time is precious on any project. Spinning your wheels in a cloud of uncertainty at the starting line creates a project that is unsafe at any speed. So what's the solution to alighting upon a list of fine, upstanding use cases upon which to base your analysis?

The answer lies in event modeling.

A brief history of event modeling

Event modeling is not new to software engineering. McMenemin and Palmer, as a way to partition large process models, first promoted event modeling in 1984. The contemporary

[1] It is not clear whether the first and last use cases are mutually exclusive.

analysis practice at the time was data flow diagramming. McMenemin and Palmer found that large projects had difficulty keeping track of transactions as they flowed through the acres of bubbles and arrows that represented the processing requirements. They were able to bring order to the apparent chaos by partitioning their models to illustrate how the system responded to specific business events.

The technique worked in practice as well as theory, and was quickly adopted by software engineers. Event modeling caught the attention of early adopters of object-orientation as well. Its focus on modeling behavior of systems with a stimulus/response paradigm fit neatly into the object-oriented view of the world. In the 1980s, Meilir Page-Jones, Steven Weiss, and Larry Constantine further formalized the discipline as part of their early work on object-oriented techniques. OCG consultants began using event modeling on the development of large-scale client/server systems in the 1990s. The results were impressive. The techniques worked! Unfortunately, the first book publication of the technique didn't occur until 1997.[2] Meanwhile, Ivar Jacobsen's 1995 book on Use Cases was causing a stir in software engineering circles, and the concept of employing a behavior-based approach to organizing requirements was taking off. Jacobsen's work closely resembled event modeling, but omitted many of the underlying formalisms. It is really a shame that these parallel efforts didn't formally merge at the outset. The rigor and heuristics of event modeling is exactly what use case modeling needs to shore up some the missteps we have witnessed in the real world.

What is an event?

Computer systems, if left to their own devices, remain inert. They are designed to leap to life only in response to external stimuli. When they do leap to life, they are programmed to behave in a predictable and repeatable manner – their behavior a reflection of the business policy they are chartered to implement. Therefore, the desired behavior of a business system can be modeled by stating how the system should respond to external *events*.

Events are stated in subject-verb-object format. Some actor does something to something, e.g., "Customer places order," "Sales Manager denies credit request," "Marketing Department changes prices." Unlike the verb-object combinations from our client's heuristically challenged use case instructor, an event must pass five tests before it is accepted onto the event list for the project:

1. An event occurs - at a specific moment in time.
2. An event occurs in the environment, not inside the system
3. The event is under the control of the environment, not the system
4. The system can detect that the event occurred
5. It is relevant, meaning that the system is chartered to do something about it.

Now we've got something solid to stand on for listing requirements from a behavioral point of view. Event modeling takes much of the guesswork out of coming up with the list of use cases.

[2] Ruble, 1997

How "low" should you go?

The nagging question of granularity plagues both event modeling and use case modeling. A use case called "Use System" is obviously far too high a level. Similarly, an event of "User clicks left mouse button" is far too low a level. The answer lies in taking the viewpoint of a businessperson. An event should represent a cohesive business transaction that completes a unit of work from the viewpoint of the event's initiator.

An event should also be technology-agnostic. For example, the event "Customer requests shipment status" could arrive at the enterprise's doorstep from a variety of technological media. The customer could request their shipments status by calling a customer service representative, who in turn, may look it up on the mainframe. They could request status via an Internet web site. They might punch their way through the Interactive Voice Response (IVR) system, or zap their way in using SMS messaging, or perhaps send an EDI transaction.

Regardless of the technology, the event's essential business policy remains unchanged. The stimulus data is the same, the processing is the same, and the response data is the same. It is the aim of event modeling to capture this essence of the business policy – and defer the design of dialog or interaction until the point where the technology is declared.

This is a significant departure from how use cases are defined in many texts. It is precisely because we spend most of our time designing *new* business systems that we adhere to the opinion that a use case that prematurely dictates an interaction design has a high likelihood of repeating the sins of the former system, and missing opportunities for business process reengineering.

By sticking to this key event modeling principle, you can avoid premature ossification of interaction design – wherein your first stab at recording requirements instead results in a potentially sub-optimal interface navigation being cast in stone. Instead, you will create an analysis specification that has more value to the business in the long run, because it can be implemented using a variety of technologies.

Advanced event modeling techniques

The discipline of event modeling includes some very useful techniques that encourage analytical discovery. First, the analyst can classify the event as to whether it is "unexpected" or "expected." Most events are "unexpected," meaning that the business (or system) never knows when a particular instance of the event will occur. "Customer places order" is a classic unexpected event in most industries.

Expected events, on the other hand, are the result of some predecessor event having established a window of expectation in the system during which a particular instance of the event is anticipated to occur. For example, "Warehouse ships order" is expected to occur based on the previous event of "Customer places order" having informed the warehouse to ship. Expected events are interesting because they can result in their failure to occur within the window of expectation, which can cause all sorts of complications in the business policy that are often overlooked by analysts during requirements gathering.

In our example of "Warehouse ships order," if we suggest that this event is "expected" to occur within a timeframe, we must ask ourselves (and the users) "At what point are we to declare the failure of the event to have occurred?" These failures, or "non-events" also pass the event litmus test because their failure can be declared to have occurred at a precise moment in time. (Perhaps in this example, a claxon should sound in the customer service department if the warehouse fails to ship an order within 5 days.)

Another peculiar type of event is the time-triggered, or temporal event. Temporal events are always expected events because they are the result of the passage of time exceeding a schedule established in the system by a predecessor event.[3]

You can see by this short introduction to event modeling, that there is a sufficient level of rigor that comes with the technique that helps the analysts create a reasonable first cut list of use cases – based on how the business is supposed to behave in response to real world events. The next step in shoring up good use case discipline is to look at how the details of a use case are written down.

Documenting Use Cases

The following section includes the best practice we have developed at OCG to ensure quality use case specification.

> **Focus on what, not how:** Use cases should focus on what the business must do to respond to events, without prescribing specific interaction design or technology. Interaction design is a complex fashioning of dialog between the user of a system, and the system itself – and is a design activity, not an analysis activity.
>
> Good interaction design takes into account the skill level of the intended user, and the power (or lack of power) of the target technology. It is often far more effectively conducted using models such as screen navigation diagrams and page & window layouts, rather than writing it out long hand.
>
> Use cases that include detailed interaction and navigational information are better suited as test scenarios than analysis documents.
>
> **Distinguish between data and processes:** One of the problems we have encountered with use case narratives is that they are often a jumble of input data, process steps and output data. Event modeling prescribes breaking out the stimulus data from the process specification from the response data. This early focus on data helps you rapidly build your data model (a.k.a. domain class model), the topic of the second paper in this series.
>
> To illustrate this point, let's look at a simple cash machine transaction, "Account Holder withdraws Funds. The stimulus is the "withdrawal request" – which is made up of a series of data elements. It is the analyst's responsibility to list the data elements, and ensure they make it onto the project's data model. Therefore, it is important for the use

[3] You will notice that the "time to" idiom slightly varies from the subject-verb-object syntax. The passage of time, however, is firmly under the control of the environment, and not the system.

case to use terminology consistent with the data model.

Processing steps are stated in a very neutral manner, avoiding the dialog interaction of the current cash machine implementation, but instead focusing on the process of making a withdrawal, regardless of whether the transaction was initiated from a cash machine, a teller, or by other electronic means.

The response data emitting from the system can have one of two outcomes in this event – either a successful withdrawal acknowledgement, or a rejection. Both responses are listed along with their corresponding data elements.

Event: <u>Account Holder withdraws funds</u>
Stimulus: Withdrawal Request:
- Account #
- PIN #,
- Account Type,
- Transaction Type,
- Transaction Amount.

Processing:
Check Account Balance
If the Account Holder's Account Balance >= Transaction Amount
    Create a Withdrawal Transaction
    Count cash
    Dispense cash
    Create a Withdrawal Acknowledgment
Otherwise, (Balance < Transaction Amount)
    Create a Withdrawal Request Rejection
(End if)

Response: (Cash) & Withdrawal Acknowledgment:
- Account #
- Transaction Date
- Transaction Amount
- Ending Balance
- Transaction ID
- Location ID
- Cash

Withdrawal Request Rejection
- Account #
- Rejection Date
- Transaction Amount
- Rejection Reason
- Location ID

You will notice the structured nature of the process specification. This leads me to my next point about sequence, selection and iteration.

Use Sequence, Selection, and Iteration to describe processes: Use cases describe business processes. Therefore, it is essential to employ the three basic constructs of process specification:

(1) Sequence: list sequential steps in the order in which they occur,
(2) Selection: Naturally occurring branches in the business logic should be documented in the main flow of the use case.
(3) Iteration: Many times, a procedure will be repeated over and over for a group of objects. Programmers refer to these as loops. In the business world, this is a naturally occurring construct and should be specifically called out when it occurs.

Use proper indentation to help the reader understand the structure of the use case. Critics of this technique claim that it makes the use case look too much like pseudocode.[4] However, my experience has shown me that the visual clarity that proper indentation brings helps lay people better comprehend a use case narrative, just as proper indentation helps a programmer comprehend someone else's code.

Avoid numbering steps. I prefer not to number my process steps. If you feel that you must number your steps, resist the temptation to refer to your steps by the literal step number. The subsequent insertion or deletion of steps can throw off your hard references, thereby creating a maintenance nightmare. Similarly, the notorious "go to "statement was derided as a poor programming practice because of difficulty of understanding the control flow of a program. The solution, as it was in good programming practices, is to reference processes specifically by name, e.g., "Check customer creditworthiness," rather than "Go to line 14.b."

Make only forward references – not backward references: There is a fervent group of use case proponents that have issued a pox on the if/then/else (selection) construct. Students of this school are taught to write use cases as if no choices are made and no exceptions occur. The basic flow, a.k.a. "Happy Day Scenario" is documented first, followed by all of the nefarious exceptions that could befall the actors if happiness eludes them that day.

I have struggled long and hard to try to make this practice work, and have come to the conclusion that it is more useful to inform the reader when a choice or branch occurs, rather than to keep them in suspense until the appendices.

The practice of eradicating if/then/else logic from a use case has some bizarre, if not unintended, consequences. The "Happy Day Scenario" is silent in regards to branch logic. However, when you read the exception flows, you are informed that "at step 4" in the basic flow, if condition x occurs, you'd come here for the exception path.[5] We are often treated to a "go to" statement to rejoin our basic flow at the end of the exception steps. This is a sheer mess.

[4] Kulak, 2000
[5] Note that the "if" statements aren't eradicated, they are simply pushed down to the end of the document.

When the time comes to consume the use case in the system's design, one must literally piece the logic branches back together. I have seen uses cases that were so fractured that all of the King's horses and men couldn't unscramble them.

The following example is a use case written using numbered steps, no branch logic, and backward references. It also contains a great deal of interaction design – wired into the use case, making it suitable only for a specific implementation. (Compare this use case specification to the event modeling style for "Account Holder withdraws funds.")

Use Case: 12. <u>Withdraw Funds</u>

Basic Flow:
1. Account Holder inserts Cash Machine Card
2. System prompts for PIN#
3. Account-Holder types PIN#
4. System validates PIN#
5. System prompts for account type (checking/savings)
6. Account-Holder indicates account type
7. System prompt for transaction type
8. Account-Holder selects "withdrawal"
9. System prompt for amount, in $5.00 increments
10. Account-Holder types amount
11. System applies withdrawal transaction to Account Holder's Account
12. System counts & dispenses cash
13. System prints Withdrawal Acknowledgment
14. System returns Cash Machine Card
15. End of Use Case

Alternative Flows:
12.1 PIN# not valid
12.1.1 At step 4 in the basic flow, PIN# is not valid
12.1.2 System displays "Wrong PIN#"
12.1.3 System prompts for PIN#
12.1.4 After 3 wrong PIN#'s, System requisitions card, Informs user to call the bank
12.1.5 End of Use Case

12.2 Insufficient Funds
12.2.1 At step 11 in the basic flow, account balance is less than withdrawal request
12.2.2 System refuses transaction
12.2.3 Resume use case at Step 14 in the basic flow

This example is small enough to seem almost manageable. However, on a real project, the numbering scheme and the backward references quickly play havoc with productivity.

There are several other tips that can help you write better use cases. They include:

- Refer to actors by their "role name" (e.g., say "Customer Service Rep" instead of "user").
- Be specific when major data entities (a.k.a. domain classes) are created, read, updated or deleted (CRUD functions).
- Factor out processes that are reused into "included use cases." These are simply internal process specifications that get referenced from multiple use cases.
- Events that change a "status" value should be charted on a state-transition model.
- Don't try to make everything fit the use case paradigm. Some problems are data-rich and process-light, and involve little or no behavior. Choose the best modeling technique for the problem at hand.

Consumption of Use Cases in good OO design

I finish this paper with a few words about consuming use cases. Bertrand Meyer, one of object-orientation's founding fathers wrote, "Use cases favor a functional approach, based on processes (actions). This approach is the reverse of OO decomposition, which focuses on data abstractions; it carries a serious risk of reverting, under the heading of object-oriented development, to the most traditional forms of functional design."[6]

What Meyer is saying is that doling out sets of use cases to different design/programming teams for implementation is a formula for disaster. Instead, each process step in a use case has to be evaluated as to its best home in an object-oriented class model. Use cases in business systems detail how the business responds to specific business events. Those events act upon many, many classes – and conversely, the same class may be acted upon by a variety of seemingly unrelated events.

A single, cohesive object-oriented design team should determine which process steps should be allocated to operations on data-centric, persistent classes, and which require new "manager" classes to orchestrate business policy. Therefore, the OO design team must manage the allocation of processes to classes, and the employment of design patterns across the entire OO architecture in order to reap the benefits of reuse and extensibility that OO promises.

Summary

Use cases have risen to prominence as the analytical format of choice for many software development organizations. As a discipline, use case modeling shares much with previous process and behavior modeling disciplines that came before. In practice, the vagueness that can accompany use case modeling can be addressed by employing the discipline of event modeling to create use cases.

Event modeling uses a technology-independent stimulus-response modeling technique to document essential business requirements, while deferring interaction design. By analyzing whether events are unexpected or expected, important policy such as the failure of an expected event to occur, and time-triggered events are often uncovered earlier in the project.

[6] Meyer, 2000

To modify a use case template to accommodate event modeling, simply separate out the stimulus data, processing steps and response data into three sections. A good use case, written in this manner will detail the input and output data elements. It will refer to the data using the same names as in the data model.

The use case steps should use sequence, selection and iteration – the three primary constructs of processing logic, to describe the business policy. I recommend against numbering use case steps, and am adamant that step numbers should not be hard-referenced by line number. The specification can contain forward references to exceptions, notes or included use cases, but no backward references.

Finally, use cases are essentially a process specification. Because any number of use cases can act upon a given business object, it is important for the designers of an object-oriented system to have a holistic view across the use cases – and therefore one should avoid creating silos of separate design teams, which will typically result in a process-oriented system that fails to meet the objectives of object-orientation.

About the author

David Ruble, a principal senior partner at Olympic Consulting Group (www.ocgworld.com), is a senior analyst, designer, author and educator. He is widely regarded as an expert in the field of business analysis, information modeling, GUI design and functional specification. He has been a principal analyst and designer of many mission-critical global corporate information systems – linking suppliers and customers worldwide. David also has significant experience designing applications in the transportation, health care and public safety sectors. His background in business, technology and art create a unique skill set that allows David to communicate with ease among business people, technologists and graphic designers.

As an educator, he has taught software engineering techniques to hundreds of students throughout the United States. He is the author of Practical Analysis & Design for Client/Server & GUI Systems, published by Prentice-Hall, 1997. His book has been used widely in colleges and universities throughout the United States and Thailand, Mexico and Argentina and is considered a timeless classic in the field of business analysis.

About Olympic Consulting Group

Olympic Consulting Group (OCG) is a full-service system architecture and development firm serving the Puget Sound region since 1997. The firm specializes in delivering high-performance consulting in the analysis, design, development and project management for complex business systems and government agencies. www.ocgworld.com

Selected Bibliography

Anderson, David. “Are Use Cases the Death of Good Design?” February 24, 2001, uidesign.net

Cockburn, Alistair. *Writing Effective Use Cases*, Addison-Wesley Pub Co; First Edition 2000

Kulak, Daryl, and Eamonn Guiney. *Use Cases: Requirements in Context*, Addison-Wesley Pub Co; 1st edition, 2000

McMenemin, S. M., and J. F. Palmer. *Essential Structured Analysis*. Englewood Cliffs, NJ: Prentice-Hall, 1984.

Meyer, Bertrand. *Object Oriented Software Construction 2nd Edition*, Englewood Cliffs, NJ: Prentice Hall PTR; 2nd edition, 2000

Page-Jones, Meilir. “Synthesis” Seminar Course Notes. Bellevue, WA: Wayland Systems, Inc., 1992

Ruble, David A. *Practical Analysis & Design for Client Server & GUI Systems*. Englewood Cliffs, NJ: Prentice-Hall, 1997

