USE CASES: THE PROS AND CONS

DONALD G. FIRESMITH

OVER THE LAST THREE YEARS, USE CASES HAVE BECOME WELL ESTABLISHED AS one of the fundamental techniques of object-oriented analysis (OOA). Although they were introduced by Ivar Jacobson to the object community at the 1987 OOPSLA conference,[1] it was the publication of his book OBJECT-ORIENTED SOFTWARE ENGINEERING: A USE CASE DRIVEN APPROACH[2] in 1992 that marked the true beginning of use cases' meteoric rise in popularity. Possibly in reaction to the previous structured methods, early object-oriented (OO) development methods overemphasized static architecture and partially ignored dynamic behavior issues during requirements analysis, especially above the individual class level where state modeling provides an important technique for dynamic behavior specification. Use cases provide a great many benefits in addition to correcting this overemphasis, and designers of most major OO development methods (including my own) have jumped on the bandwagon and added use cases during the last few years. In the resulting hoopla and hype, however, there has been little discussion of the limitations and potential pitfalls associated with use cases. In this column I attempt to provide a more balanced presentation and to caution against the uncritical acceptance of use cases as the latest patent medicine for all software ailments.

DEFINITIONS

The term *use case* was introduced by Ivar Jacobson et al.[1] and has been defined.[2–4] A use case is a description of a cohesive set of possible dialogues (i.e., series* of interactions) that an individual actor initiates with a system.

*A use case typically involves branching or looping and may depend on the state of the system and any parameters of the interactions between actors and the system.

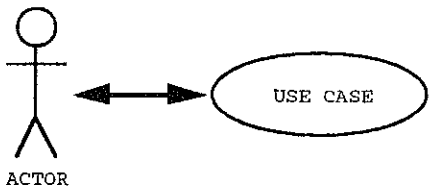

FIGURE 1. *The primary use case notations.*

An actor is a role played by a user (i.e., an external entity that interacts directly with the system) (Figure 1). A use case is thus a general way of using some part of the functionality of a system.

TIP

A use case is not a single scenario.

A use case is not a single scenario but rather a "class" that specifies a set of related usage scenarios, each of which captures a specific course of interactions that take place between one or more actors and the system. Therefore, the description of an individual use case typically can be divided into a *basic course* and zero or more *alternative courses*. The basic course of a use case is the most common or important sequence of transactions that satisfy the use case. The basic course is therefore always developed first. The alternative courses are variants of the basic course and are often used to identify error handling. Within reason, the more alternative courses identified and described, the more complete the description of the use case and the more robust the resulting system.

As a user-centered analysis technique, the purpose of a use case is to yield a result of measurable value to an actor in response to the initial request of that actor. A use case may involve multiple actors, but only a single actor initiates the use case. Because actors are beyond the scope of the system, use-case modeling ignores direct interactions between actors.

A use case may be an *abstract use case* or a *concrete use case*. An abstract use case will not be instantiated on its own but is only meaningful when used to describe functionality that is common between other use cases. On the other hand, a concrete use case can be instantiated to create a specific scenario.

According to Ivar Jacobson, use cases are related by two main associations: *extends* and *uses*. The extend association specifies how one use-case description inserts itself into, and thus extends, a second use-case description that is completely independent and ignorant of the first use case. Depending on some condition, the second use case may be performed either with or with-

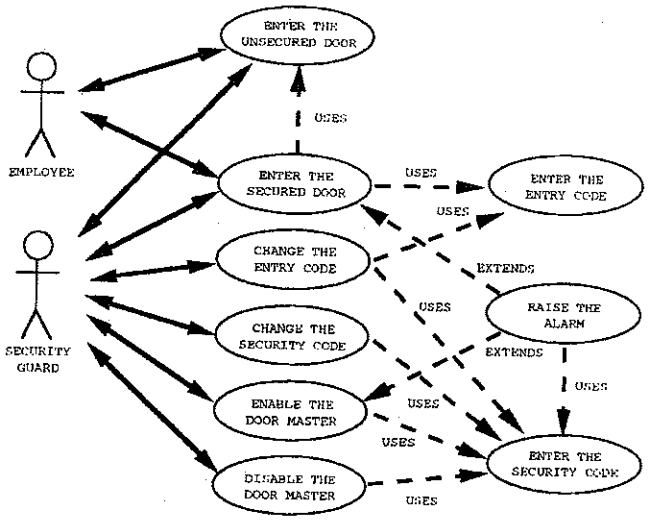

FIGURE 2. *An example use-case model.*

TIP

Extends are a kind of "inheritance."

out the extending use case. Extends can therefore be viewed as a kind of "inheritance" between use cases in which the original use case definition is extended by the extending use case description to form a new "combined" use case. On the other hand, the uses association can be viewed as a kind of "delegation" or "aggregation" that captures how one or more use-case descriptions incorporate the common description of another use case. These two associations are closely related and easy to confuse. One clue as to which is which is that if A extends B, then the extended B "contains" A, whereas if A uses B, then A "calls" B. The actual distinction between these two associations is unclear, and Rumbaugh[4] has thankfully combined them into a single *adds* association from the main concrete use case to the abstract use cases that it uses.

Clearly, use cases are functional abstractions and are thus large operations, the implementations of which thread through multiple objects and classes. However, a use case need not have anything to do with objects. As pointed out by Jacobson,[5] "it should be clear that use-case modeling is a technique that is quite independent of object modeling. Use-case modeling can be applied to any methodology—structured or object-oriented. It is a discipline of its own, orthogonal to object modeling."

AN EXAMPLE

The requirements for Door Master, a security system for controlling entry of employees through a secured door, are documented in a ROAD column.[6] Except for those requirements concerned with initialization, the functional requirements for Door Master are captured in the following nine use cases:

1. ENTER_THE_DISABLED_DOOR: Employees and security guards enter freely through the door when Door Master is disabled.
2. ENTER_THE_SECURED_DOOR: Employees and security guards enter through the door by (1) entering the entry code on the numeric keypad, (2) entering through the door, and (3) closing the door behind them.
3. CHANGE_THE_ENTRY_CODE: Security guards change the entry code by (1) pressing the "change entry code" button on the control panel, (2) providing authorization by entering the security code on the numeric keypad, (3) entering the new entry code on the numeric keypad, and (4) verifying the new entry code by reentering it on the numeric keypad.
4. CHANGE_THE_SECURITY_CODE: Security guards change the security code by (1) pressing the "change security code" button on the control panel, (2) providing authorization by entering the old security code on the numeric keypad, (3) entering the new security code on the numeric keypad, and (4) verifying the new security code by reentering it on the numeric keypad.
5. ENABLE_THE_DOOR_MASTER: Security guards enable Door Master by (1) pressing the "enable" button on the control panel and (2) providing authorization by entering the security code on the numeric keypad. Door Master then (3) turns off the disabled light, (4) turns on the enabled light, and (5) locks the door.
6. DISABLE_THE_DOOR_MASTER: Security guards disable Door Master by (1) pressing the "disable" button on the control panel and (2) providing authorization by entering the security code on the numeric keypad. Door Master then (3) turns off the enabled light, (4) turns on the disabled light, and (5) unlocks the door.

The following two abstract use cases are common to, and are therefore used by, five of the concrete use cases:

7. ENTER_THE_ENTRY_CODE: Employees and security guards enter the entry code by pressing five keys on the numeric keypad followed by the "enter" key. Door Master beeps after each key and verifies the entry code.
8. ENTER_THE_SECURITY_CODE: Employees and security guards enter

the entry code by pressing seven keys on the numeric keypad followed by the "enter" key. Door Master beeps after each key and verifies the entry code.

The following abstract use case extends the ENABLE_THE_DOOR_MASTER and ENTER_THE_SECURED_DOOR use cases:

9. RAISE_THE_ALARM: The alarm is raised if the door is left open too long or if the door is not shut when Door Master is enabled. The security guards disable the alarm by entering the security code.

The Benefits of Use Cases

Use cases have become extremely popular since the publication of Object-Oriented Software Engineering: A Use Case Driven Approach in 1992. They have been added to numerous OO development methods (e.g., Booch, Firesmith, Rumbaugh) because they offer many important advantages, including the following:

- As a user-centered technique, use cases help ensure that the correct system is developed by capturing the requirements from the user's point of view.
- Use cases are a powerful technique for the elicitation and documentation of *blackbox functional* requirements.
- Because they are written in natural language, use cases are easy to understand and provide an excellent way for communicating with customers and users. Although computer-aided software engineering (CASE) tools are useful for drawing the corresponding interaction diagrams, use cases themselves *require* remarkably little tool support.
- Use cases can help manage the complexity of large projects by decomposing the problem into major functions (i.e., use cases) and by specifying applications from the users' perspective.
- Because they typically involve the collaboration of multiple objects and classes, use cases help provide the rationale for the messages that glue the objects and classes together. Use cases also provide an alternative to the *overemphasis* of traditional OO development methods on such static architecture issues as inheritance and the identification of objects and classes.
- Use cases have emphasized the use of lower-level scenarios, thereby indirectly supporting Booch's important concept of a *mechanism*, a

kind of pattern that captures how "objects collaborate to provide some behavior that satisfies a requirement of the problem."[7]

- Use cases provide a good basis for the verification of the higher-level models (via role-playing) and for the validation of the functional requirements (via acceptance testing).*
- Use cases provide an objective means of project-tracking in which earned value can be defined in terms of use cases implemented, tested, and delivered.
- Use cases can form the foundation on which to specify end-to-end timing requirements for real-time applications.

THE DANGERS OF MISUSING USE CASES

Because of their many important advantages and extreme popularity, use cases have become a fundamental part of object technology and have been incorporated in one form or another into most major OO development methods. In the rush to jump onto the use-case bandwagon, use cases have been perceived by some as either a panacea or as an end in and of themselves. Unfortunately, this has often led to the uncritical acceptance of use cases without any examination of their numerous limitations and ample opportunities they offer for misuse. The following provides an overview of the major risks associated with use cases:

- Use cases are *not* object oriented. Each use case captures a major functional abstraction that can cause the numerous problems with functional decomposition that object technology was to avoid. These problems include:
- The functional nature of use cases naturally leads to the functional decomposition of a system in terms of concrete and abstract use cases that are related by extends and uses associations. Each individual use case involves different features of multiple objects and classes, and each individual object or class is often involved in the implementation of multiple use cases. Therefore, any decomposition based on use cases scatters the features of the objects and classes among the individual use cases. On large projects, different use cases are often assigned to different teams of developers or to different builds and releases. Because the use cases do not map one-to-one to the objects and classes, these teams

* Because a use case (class) is not as specific as a usage scenario (instance), use cases may lack sufficient formality and detail to supply adequate criteria for the passing of acceptance tests.

can easily design and code multiple, redundant, partial variants of the same classes, producing a corresponding decrease in productivity, reuse, and maintainability. This scattering of objects to use cases leads to the Humpty Dumpty effect, in which all the king's designers and all the king's coders are unlikely to put the objects and classes back together again without a massive expenditure of time and effort.

TIP

Note the Humpty Dumpty effect.

- The use-case model and the object model belong to different paradigms (i.e., functional and OO) and therefore use different concepts, terminology, techniques, and notations. The simple structure of the use-case model does not clearly map to the network structure of the object model with its collaborating objects and classes. The requirements trace from the use cases to the objects and classes is also not one-to-one. These mappings are informal and somewhat arbitrary, providing little guidance to the designer as to the identification of objects, classes, and their interactions. The situation is clearly reminiscent of the large semantic gap that existed between the data flow diagrams (network) of structured analysis and the structure charts (hierarchy) of structured design. The use of the single object paradigm was supposed to avoid this problem.

TIP

The use-case model and the object model belong to different paradigms.

- Another potential problem with use case modeling is knowing when to stop. When one is building a nontrivial application, there are often a great number of use cases that can produce an essentially infinite number of usage scenarios, especially with today's graphical user interfaces and event-driven systems. How many use cases are required to adequately specify a nontrivial, real-world application? As object technology is applied to ever increasingly complex projects, the simple examples and techniques of the textbooks often have trouble scaling up. The use of concurrency and distributed architectures often means that the order of the interactions between the system and its environment is potentially infinite. Too few use cases result in an inadequate specification, while too many use cases lead to functional decomposition and the scattering of objects and classes to the four winds. Often, systems and software engineers must limit their analysis to the most obvious or important scenarios and hope that their analysis generalizes to all use cases.
- Although use cases are functional abstractions, use-case modeling typically does not yet apply all of the traditional techniques that are useful

for analyzing and designing functional abstractions. Most current techniques do not easily handle the existence of branches and loops in the logic of a use case. Interaction diagrams are primarily oriented towards a simple, linear sequence of interactions between the actors and the major classes of the system. The use of abstract use cases and either extends or uses associations to solve this problem only exacerbates the functional decomposition problem. Some approach similar to that of the basis paths of structured testing would clearly help determine the adequacy of the use case model, but such an approach is not yet available to the typical developer. Most techniques do not address the issues of concurrency and the different types of messages that result. As illustrated in Rumbaugh[4] and Firesmith,[6] the concepts of preconditions, postconditions, invariants, and triggers should also be added to better analyze and specify use cases.

- Since they are created at the highest level of abstraction before objects and classes have been identified, use cases ignore the encapsulation of attributes and operations into objects. Use cases therefore typically ignore issues of state modeling that clearly impact the applicability of some use cases. Any required ordering of use cases is ignored and should be captured using some variation of Firesmith's scenario lifecycle[6] or Fusion's event lifecycle.[8] The basic ideas and techniques of use cases should also be applied to Booch mechanisms[7] and integration testing, but adequate extensions have yet to be published.
- Another major problem with use-case modeling is the lack of formality in the definitions of the terms *use case*, *actor*, *extends*, and *uses*. Similarly, the specification of individual use cases in natural languages such as English provides ample room for miscommunication and misunderstandings. Use cases provide a much less formal specification of their instances (i.e., individual usage scenarios) than do classes of objects. Whereas the inheritance relationship between classes of objects is well defined and has been automated by compilers, the inheritance and delegation relationships provided by extends and uses associations are much less well defined. While everything may seem clear at the highest level of abstraction, the translation of use cases into design and code at lower levels of abstraction is based on informal human understanding of what must be done. This also causes problems when it comes to using use cases for the specification of acceptance tests because the criteria for passing those tests may not be adequately defined.
- Another major problem is the archetypal subsystem architecture that can result from blindly using use cases. Several examples in books and papers have consisted of a single functional control object representing the logic of an individual use case and several dumb entity objects

controlled by the controller object. They also may have included an interface object for each actor involved with the use case. Such an architecture typically exhibits poor encapsulation, excessive coupling, and an inadequate distribution of the intelligence of the application among the classes. Such architectures are less maintainable than more object-oriented architectures.

- Use cases are defined in terms of interactions between one or more actors and the system to be developed. However, all systems do not have actors, and systems may include signification functionality that is not a reaction to an actor's input. Embedded systems may perform major control functions without significant user input. Concurrent objects and classes need not passively wait for incoming messages to react. They may instead proactively make decisions based on results derived from polling terminators. Traditional use-case modeling seems less appropriate for such applications.
- Finally, the use of use cases as the foundation of incremental development and project tracking has its limitations. Basing increments on functional use cases threatens to cause the same problems with basing builds on major system functions. Instead of building complete classes, developers will tend to create partial variants that require more iteration from build to build than is necessary. In turn, this will unnecessarily increase the maintenance costs of inheritance hierarchies. Basing earned value on the number of use cases implemented may be misleading because all use cases may not be of equal value to the user and because of the previously mentioned problems due to functional decomposition and the scattering of partial variant objects and classes among use cases.

Conclusion

What, then, should developers do? Use cases clearly offer many important benefits and are powerful weapons that probably should be in the arsenal of all software analysts, designers, and testers. Unfortunately, however, they are functional rather than object-oriented and can significantly compromise the benefits of object technology if blindly added to the OO development process. Fortunately, the risks associated with use-case modeling can be mitigated through knowledge, training, and avoiding an overenthusiastic acceptance. Use cases should be only one of several ways of capturing user requirements. Models of objects, classes, and their semantic relationships should be consistent with, but not totally driven by, the use cases. Designers should beware of and minimize scattering the features of a use case's objects and classes, and they should exercise great care to avoid the creation of partial, redundant variants

of classes, especially on large projects involving multiple builds and releases. The architectural guidelines of Rebecca Wirfs-Brock[9] should be followed to avoid creating excessive functional controller objects that dictate the behavior of dumb entity objects. Most importantly, use cases should not be used as an excuse to revert to the bad old days of functional decomposition and functionally decomposed requirements specifications.

References

1. Jacobson, I. Object-oriented development in an industrial environment, SIGPLAN NOTICES 22(12):183–191.
2. Jacobson, I., M. Christersson, P. Jonsson, and G. Overgaard. OBJECT-ORIENTED SOFTWARE ENGINEERING: A USE CASE DRIVEN APPROACH, Addison-Wesley, Wokingham, UK, 1992.
3. Jacobson, I., M. Ericsson, and A. Jacobson. THE OBJECT ADVANTAGE: BUSINESS PROCESS RE-ENGINEERING WITH OBJECT TECHNOLOGY, Addison-Wesley, Wokingham, UK, 1995.
4. Rumbaugh, J. Getting started: Using use cases to capture requirements, JOURNAL OF OBJECT-ORIENTED PROGRAMMING 7(5):8–12, 1994.
5. Jacobson, I. Basic use-case modeling (continued), REPORT ON OBJECT-ORIENTED ANALYSIS AND DESIGN 1(3):7–9, 1994.
6. Firesmith, D. Modeling the dynamic behavior of systems, mechanisms, and classes with scenarios, REPORT ON OBJECT-ORIENTED ANALYSIS AND DESIGN 1(2):32–36, 1994.
7. Booch, G. OBJECT-ORIENTED ANALYSIS AND DESIGN WITH APPLICATIONS, Benjamin/Cummings, Redwood City, CA, 1994.
8. Coleman, D., et al. OBJECT-ORIENTED DEVELOPMENT: THE FUSION METHOD, Prentice Hall, Englewood Cliffs, NJ, 1994.
9. Wirfs-Brock, R., B. Wilkerson, and L. Wiener. DESIGNING OBJECT-ORIENTED SOFTWARE, Prentice Hall, Englewood Cliffs, NJ, 1990.