Use cases-basics
Kumar Vadaparty
Journal of Object - Oriented Programming; Feb 2000; 12, 9; ProQuest Computing
pg. 4

UML & BEYOND

Kumar Vadaparty

Use Cases—Basics

Welcome to UML and Beyond. The goal of this column is twofold:

- To introduce the new and exciting confluence of software modeling techniques that the Unified Modeling Language (UML) gives us.
- To formalize the notion of *successive refinement* of deliverables in the software life cycle, and show how UML, possibly with customizations, can be used to *represent* and trace successive refinements of deliverables.

The typical reader is not expected to be familiar with UML in particular or any modeling languages in general; experience in different stages of the software development life cycle (from requirements gathering to deployment) is sufficient, but specialists, too, will gain a lot from this series: customizing UML to represent certain refinements, development process and tool support for UML, etc.

The objective of UML is to help two communities: *architects* and software modeling *tool vendors.*

Architects: To help architects specify, in a human-readable visual language with precise syntax and semantics, the artifacts of the software development life cycle. The UML document says:

> *UML V1.3 alpha R5 March 1999: The Unified Modeling Language(UML) provides system architects working on object analysis and design with one consistent language for specifying, visualizing, constructing, and documenting the artifacts of software systems, as well as for business modeling . . . a human-readable notation for representing OA&D models. This document defines the UML notation, an elegant graphic syntax for consistently expressing the UML's rich semantics.*

Notation is an essential part of OA&D modeling and the UML.

Tool vendors: To provide tool vendors with a common substrate to store different modeling artifacts so that an artifact developed by one tool can be viewed and updated by another (again, from the UML document):

> *UML V1.3 alpha R5 March 1999: One of the primary goals of UML is to advance the state of the industry by enabling object visual modeling tool interoperability [by providing] formal definition of a common object analysis and design (OA&D) metamodel to represent the semantics of OA&D models [and] IDL specifications for mechanisms for model interchange between OA&D tools.*

Anyone who has built any software system must have drawn sketches describing the main principles behind the system before, during, or after building it. *Every such activity* is a part of modeling. Over the last several years there have been many efforts to formalize the graphical language used in these software-modeling sketches. What UML provides is a unified, industry-standard, graphical language that enables one to develop software models. A brief history of the emergence of this confluence from its different ancestral streams is given by Rumbaugh.[1]

Formal visual specifications also enable one to discover and reuse frequently used patterns. Such reuse provides both cost-effective implementation (eliminating duplicative efforts) and standardized development across the enterprise.

COLUMN EMPHASIS

The emphasis of this column is *not* a comprehensive description of UML; OMG's UML Specification serves that purpose (www.rational.com/uml). *Rather, the emphasis of this series is to discuss how the UML can be effectively used for modeling different artifacts in the software development life cycle and to go beyond by considering issues that are not in the scope of UML as well* (e.g., reuse and process related issues). The following are some of the main goals. Many of these topics will be covered in depth in future columns, so readers who do not understand the material in these descriptions need not be concerned.

Successive refinement of deliverables

There is considerable discussion in the literature on how to depict requirements using use cases, but the transition from use cases to components is not adequately presented. We believe the most important step after detailing use cases is *identifying tier-aware self-deployable logical components with well-defined interfaces;* then comes the detailed design and implementation of these components (possibly using classes where appropriate) or the purchase

Dr. Kumar Vadaparty works as a Technical Manager for a major client/server project at Merrill-Lynch. He can be contacted at kumar_vadaparty@ml.com.

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

of custom-developed off-the-shelf products if available and affordable. We will discuss this issue in depth in the coming months.

Reuse

UML enables us to discuss reuse of software entities in a formal manner; it provides the vocabulary and the notation to represent reusable parts and their interrelationships. This topic is not addressed explicitly in the UML specification document because it belongs to "Process"—a topic that is out of scope for UML. However, this is a very important topic for robust software development, and we will discuss this in depth.

Reuse happens at different levels. At the implementation level it is referred to through "idioms"[2] in the context of C++ implementation (idioms are language-specific). At the design level, these idioms are abstracted into language-independent patterns[3,4] with no reliance on specific languages, but the reusability is still at the class level. Some reusable design patterns (especially delegation) can be further elevated into reusable patterns between components. Component patterns (called Frameworks[5,6]) provide a reusable solution at a much higher level of abstraction (and thus a larger grain of reuse). Finally, higher-order Frameworks or architecture patterns provide reuse at the entire architecture level.[6]

We will touch upon these matters and show how the UML can be quite useful in discovering and depicting reusable architectures.

Guidelines for choosing the right formalism

We will illustrate how to identify the right formalism to depict real-world situations, and conversely, how to recognize typical real-world problems that one would model using a given formalism. We draw from the user guide[7] and go further. For example, we depict some important multitiered architectural patterns (appropriate for the Web) and show how they constitute very useful UML idioms.

Process Discussions

UML does *not* prescribe a development process or method. However, it is very difficult to discuss software artifacts in isolation without discussing how they would evolve in a software life cycle—that is, we need the context of a process. Therefore, invariably, some of our discussions will land in the area of process. In such discussions, we refer to well-known processes[5,8] and their adaptations. Our interest in process is primarily towards *deliverable representation* and *refinement* (evolution). We do *not* touch many other aspects of process, e.g., how to allocate work to different roles, how the workflow among different "workers" flows, etc. We concentrate on *what*, not *who*.

Traceability of deliverable refinement

UML does *not* specify rules to connect deliverables using hyperlinks. It states that (i) hyperlinking different deliverables is a task that belongs to tool vendors and (ii) we do not know enough about how to hyperlink to make this a standard.* We agree with this position. Indeed, we would like to use this column as a sounding board for discussing different ways to link deliverables, and we hope that the tool vendors will pick from the results of these discussions. Hyperlinks help us trace the evolution of deliverables (from requirements to code) through the software life cycle. A discussion on ways to provide such traceability is of great value to architects.

Clarify terminology

There is considerable confusion in the literature about the terms "objects," "classes," "components," "object-oriented versus component-based" (implying that somehow one supplants the other), and so on. The best description of such confusion is given in *Component Software Beyond Object-Oriented Programming*[6]:

> *The terms "component" and "object" are often used interchangeably. In addition, constructions such as "component object" are used. Objects are said to be instances of classes or clones of prototype objects. Objects and components are both making their services available through interfaces, interfaces are of certain types or categories. As if this was not enough, object and component interactions are described using object and component patterns and prescribed using object and component frameworks. Both components and frameworks are said to be whitebox or blackbox ...*

Given all the confusion, it is actually a surprise that any two programmers can communicate; most of such communication remains at the implementation level where there is no confusion—"COM component" means the same for every one (as does JavaBean, etc.). We would like to clarify some of these confusions and provide clear one-to-one textual descriptions for the UML artifacts we develop.

Reviewing tools

We will review some of the tools on the market that facilitate using UML to model the software life cycle, e.g., Rational, Platinum (now Computer Associates), etc.

USE CASES

The UML specification document does *not* mandate or endorse a specific software development process but does mention the importance of use cases in the software development life cycle ("use-case driven"):

> *The UML is intentionally process independent, and defining a standard process was not a goal of the UML or OMG's RFP...[However] its developers have recognized the value of a* use-case driven, architecture-centric, iterative, and incremental process, *so were careful to enable (but not require) this with the UML.*

*3-8 UML V1.3 alpha R5 March 1999 (section 3.4):
A notation on a piece of paper contains no hidden information. A notation on a computer screen may contain additional invisible hyperlinks that are not apparent in a static view, but that can be invoked dynamically to access some other piece of information, either in a graphical view or in a textual table. Such dynamic links are as much a part of a dynamic notation as the visible information, but this guide does not prescribe their form. We regard them as a tool responsibility.... Eventually some of the dynamic notations may become well enough established to standardize them, but we do not feel that we should do so now.

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

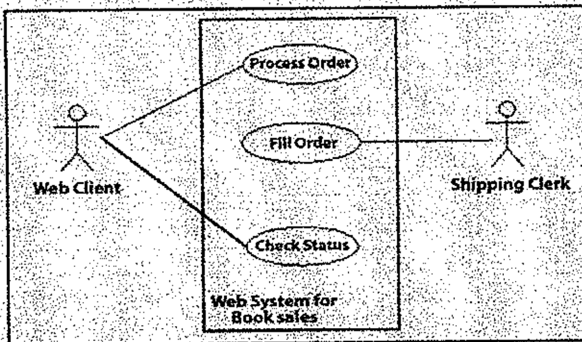

Figure 1. Use case diagram for purchase of books on the Web (simplified version).

We start our discussion of UML with illustrations of use cases and their importance to architects, managers, and users. Let us consider a Web-based bookshop (e.g., Amazon.com) where a Web client purchases books on the Web, and a shipping clerk fills in the order and ships the book. Figure 1's use case diagram captures this at a certain level of abstraction.

First, let us understand the diagram (although it is quite easy to follow, which is the merit of use case diagrams). Each ellipse with a horizontal major axis is called a "use case," and each stick figure is called an "actor."

- *Use cases*: A use case *encapsulates* or abstracts *a coherent unit of service*, functionality, or action to be provided by self-identifiable parts of the system being analyzed. Thus, it is a "server" to the outside "clients" (actors) that act on it.

  UML V1.3 alpha R5 March 1999: A use case is a coherent unit of functionality provided by a system, a subsystem, or a class as manifested by sequences of messages exchanged among the system and one or more outside interactors (called actors) together with actions performed by the system.

- Actor: An actor represents an agent that interacts with the system but functions outside the scope of our analysis. Actors circumscribe the boundaries of the system under investigation. A more detailed refinement of a particular use case (as if it is a system in its own right) may, in turn, have other actors and use cases. An actor could be a human or an external system outside the scope of current analysis.

  UML V1.3 alpha R5 March 1999: An actor defines a coherent set of roles that users of an entity can play when interacting with the entity. An actor has one role for each use case it communicates with.

The use case diagram in Figure 1 captures the following: There are two client-types in this system—the Web client and the shipping clerk. Both of them happen to be human actors (although this is not necessary in general). The Web client issues an order to the system, and the subsystem, represented by the use case ProcessOrder, receives it and processes it. The subsystem, represented by CheckStatus, enables the Web client to check the status of a previous order. Finally, the subsystem, represented by the use case FillOrder, enables filling in a shipped order.

Thus, decomposing a system into use cases and actors means, essentially, decomposing a system into service providers and service users or clients. The first question that comes to mind is "*How can anyone advance software engineering principles by a bunch of ovals and stick figures?* You must be kidding." If we leave our requirements gathering right here, then indeed we reap little benefit out of these so-called use cases. Several future installments will add more detail to this simple-looking formalism, and we assure our readers that they will be more than convinced of their value; so much so, I hope, that they will not proceed with building a new system without a use case diagram.

HOW DOES ONE "BREAK" A SYSTEM INTO OVALS AND STICK FIGURES?

An important question: What are the criteria to determine actors and use cases?

Recognizing actors

The following are some of the guidelines for recognizing actors within a system:

- *Human users*: Entities that are completely outside the domain of software development, such as human users, are almost invariably represented as actors.
- *External data providers and data consumers*: An external data feed to the system being analyzed will be represented as an actor. For example, data feeds such as Reuters and AP are represented as actors when analyzing WallStreetJournal or NYTimes. Conversely, an external data consumer of a system will be represented as an actor. That is, consumers such as NYTimes and WashingtonPost will be represented as actors when analyzing systems such as AP.
- *Legacy System Feeds*: The notion of "external" is relative. It does *not* have to be a different company. Those databases that are outside the scope of the system being analyzed, but are populated by "legacy" (i.e., systems developed prior to the system being currently analyzed) will also be represented as actors.
- *Off-the-shelf third-party products*: The system being analyzed may use some off-the-shelf third-party products of which we know only external behavior and nothing about implementation. Clearly such entities will be represented as actors.
- *Enterprise-wide system decomposition*: Finally, strictly with an intent to divide labor across different organizational structures within a large project, entities (i.e., modules, programs, etc.) that are outside the organizational structure of the system being analyzed will be represented as actors.

Chopping a system up into use cases

Decomposing the system into use cases is by far the most difficult task. For example, in Figure 1 there is no reason why we could not have combined all three use cases into one use case, say, TheSystem, which will perform all three main activities: processing, filling, and checking the status of an order. Just separating them into three different use cases *does not* make the use

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

case diagram inherently better *or does it?* Also, let us consider the symmetric variant of the question: why can't we break the use case ProcessOrder into AuthenticateUser, ReceiveOrder, and ProcessOrder? Is the use case decomposition shown in Figure 1 *inherently inferior* to the one in which ProcessOrder is broken into the three use cases?

There are a number of factors that affect the decision of where to draw the line. As can be expected, this is not a black and white decision but involves considerable engineering ingenuity and software development experience. Here we gather some of the important factors:

- *Encapsulation and elegance (rooted in reusability)*: From experience in OO analysis, one can easily see that Process Order should be separate from StatusChecking. After all, (i) they both represent coherent but distinct business functions, and (ii) there could be many other subsystems (use cases) that might want to interact with StatusChecking. Therefore, it is appropriate to represent them as separate use cases. However, continuing with the same logic, we should then break ProcessOrder into three use cases: AuthenticateUser, ReceiveOrder, and ProcessOrder. After all, authentication can happen for many other use cases as well. Thus, encapsulation and reusability provide a force in the direction of "breaking" into further smaller use cases.

- *Organizational structure of system users*: Suppose the entire "Web Bookshop" that we are considering is a mom-and-pop store, run from a basement. We definitely do not need even the level of decomposition in Figure 1. We could easily have combined all of the use cases into one big TheSystem use case with individual use cases as tasks of the original use case.

  On the contrary, if it is a large enterprise (e.g., barnesandnoble.com or Amazon.com), then the decomposition is appropriate. This is because the people who fill in orders are organizationally separate and have a clear idea of how the subsystem corresponding to FillOrders is supposed to work. In fact, one may even want to detach AuthenticateUser as a separate use case from ProcessOrders. This is because the requirements for AuthenticateUser may very well belong to a whole different subgroup of users (the security people) who police the overall firm-wide authentication.

- *Architectural affinity*: Recall that use cases are service providers. Suppose that parts of a use case are to be eventually running on very different architectural boundaries. Then it is appropriate to break them cleanly into separate use cases even if functionally we capture them together. In our example in Figure 1, assume that after receiving the order for a book, the system then has to send it to a remote back-end server that in turn goes through a certain workflow (e.g., see if the book is available, contact vendors if the book is not available). The middle layer simply confirms the receipt of the order.[†] In such situations, it is quite appropriate to break ProcessOrder into two separate use cases, ReceiveandConfirmOrder and ProcessOrder.

These considerations give us a first cut at how to decompose a system into use cases and actors. In summary, we point out the following: Decomposing a system into use cases has to be done with the explicit cognizance of (i) where that use case will "end up" (i.e., tier awareness and architectural affinity), (ii) who will provide the requirements (i.e., user organizational structure), and (iii) how to exploit reuse and gain potential for accommodating future changes (i.e., elegance and encapsulation).

Some people tend to argue in favor of "pristine, architecture-independent" structuring of the system into use cases, saying that real-world constraints such as architecture stifle the creativity involved in decomposing the system into use cases. My own experience and the motivating suggestions of others[5,6,8,9] indicates that it is more creative and useful to come up with use case decomposition while keeping in mind the real-world constraints. Moreover, pretending to develop from a clean slate and then force-fitting real-world constraints retroactively (after all, the real world is not going to go away) is a compromise at best. Certain awareness of the final home of the use cases must play an important role in the decomposition. There are, of course, more questions to be answered:

- How do we represent communication between use cases and actors, and between use cases themselves (is it possible, and if so, how?)? Do they have interfaces?
- How do we represent the flow of data back and forth from use case to actor and vice versa?
- Use cases seem to be representing "Islands of Behavior." How do we connect them? For example, ProcessOrder and CheckStatus must be connected some way. How? Is there such a notion as an "end-to-end use case of a system"?

These and many other questions will be addressed in the future columns. Stay tuned! ■

References

1. Rumbaugh, J., I. Jacobson, and G. Booch. *The Unified Modeling Language Reference Manual*, Addison–Wesley, Reading, MA, 1998.
2. Coplien, J. *Advanced C++ Programming Styles and Idioms*, Addison–Wesley, Reading, MA, 1992.
3. Gamma, E. et al. *Design Patterns: Elements of Reusable Object-Oriented Software*, Addison–Wesley, Reading, MA, 1995.
4. Jacobson, I., M. Griss, and P. Jonsson. *Software Reuse*, Addison–Wesley, Reading, MA, 1998.
5. D'Souza, D. and A. Wills. *Objects, Components and Frameworks with UML: The Catalysis Approach*, Addison–Wesley, Reading, MA, 1998.
6. Szyperski, C. *Component Software: Beyond Object-Oriented Programming*, Addison–Wesley, Reading, MA, 1998.
7. Booch, G., J. Rumbaugh, and I. Jacobson. *The Unified Modeling Language User Guide*, Addison–Wesley, Reading, MA, 1998.
8. Jacobson, I., J. Rumbaugh, and G. Booch. *The Unified Software Development Process*, Addison–Wesley, Reading, MA, 1998.
9. Schneider, G. and J. Winters. *Applying Use Cases: A Practical Guide*, Addison–Wesley, Reading, MA, 1998

Quantity reprints of this article can be purchased by phone: 717.399.1900, ext. 139 or by email: sales@rmsreprints.com.

[†] This is similar to Amazon.com. When we order back-ordered books, we simply get a confirming e-mail first, and when the order is processed later (perhaps within the same day or later), we get another e-mail.

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.