Use Case Concepts from an RM-ODP Perspective

Guy Genilloud[(1)], William F. Frank[(2)]

[(1)] Guy Genilloud, Departamento de Informatica, Escuela Politecnica Superior de Leganes, Universidad Carlos III de Madrid
[(2)] X-Change Technologies Group LLC36 West 44th St., Suite 1201, New York, NY 10036
[(1)] guy.genilloud@ie.inf.uc3m.es, [(2)] wff@xchangetechnologies.com

Abstract. Use Cases have achieved wide use in software engineering. However, there is still controversy concerning the semantics of use case models; some practitioners find the official UML doctrines very awkward. The source of this awkwardness is the failure of UML to identify what a use case model is a model of – the official doctrine is self-referential, in that use cases are seen as kinds of specifications, rather than kinds of things in the world. The Extend relationship, in particular, illustrates these problems. In this paper, we show how the terminology and explanations of use cases in the UML standard(s) can confuse learners and users of use case techniques. We then show how the ontology of the RM-ODP can be used to analyze the problem, communicate effectively about it, and find appropriate solutions.

1. Introduction

This paper is addressed to experts of the UML and Use Case communities. We hope that it will give them a better understanding of the problem that they face in communicating to demanding students and an even more demanding future, and will be a step toward reaching some common solutions to these problems.

Use Cases have achieved wide use in software engineering for specifying the observable behavior of systems. However, there is still controversy among practitioners about when and how to use some features of use case modeling. One feature that has created considerable difficulty is the Extend relationship. We believe that these difficulties are an outcome of the lack of a firm logical foundation for use case modeling – a clear ontology both of the things modeled by and found in a use case model. This persistent confusion over Use Case concepts techniques may explain the persistent lack of good tools supporting both the textual and the graphical aspects of Use Case specifications.

In Section 2, we show that important root causes of the problem can be traced to UML's unnatural ontology, which feeds confusion between the concepts of use case, use case instance and use case type. In Section 3, we present the ontology of the RM-ODP, in particular the way it handles instances, specifications and types. In Section 4, we present the use case concepts from an RM-ODP perspective.

2. Use Cases and the Ontology of UML

We use "ontology" in the classical sense of the recognition of the categories of being that exist in a language or system of thought, and the relationships among them. The specifiers of UML have steered clear of the fundamental issues of what UML models are models of, and what a UML model itself is, as a category of being. As a result, the explanations given for the semantics of UML in general, and use cases in particular, are not, on any but a cursory and uncritical acceptance, coherent.

2.1. The Strange Ontology of UML and its Application to Use Cases

The UML terminology is such that the most usual terms denote either types or specifications of things, rather than the things themselves. Another, entirely separate term must be used for denoting individuals that are instances of those types (or specifications). For example, UML 1.5 [1] defines "message" as "*A specification of the conveyance of information…*" but also speaks of "*sending a message*" or of "*ordered messages*." But "sending a specification" or "ordered specifications" is not what is meant. The fact that UML does not always use "stimulus" (the defined term for an instance of a message) in these cases shows how impractical it is to choose different terms for the instance and the type. This way of thinking is not applied in human languages. For example, imagine that individual dogs were called 'goodles', because the word 'dog' was reserved for the type or concept of a dog. People would have to say: I saw a Dog instance the other day – that goodle was wagging his tail.

In UML 2, Use Case is defined as follows:

"A use case is the specification of a set of actions performed by a system, which yields an observable result that is, typically, of value for one or more actors or other stakeholders of the system." [2].

UML-2 gives the following precision to its readers:

Strictly speaking, the term "use case" refers to a use case type. An instance of a use case refers to an occurrence of the emergent behavior that conforms to the corresponding use case type." [2].

This is a step in the right direction, since UML 2 makes a clear distinction between things outside the model, that are to be modeled, like behaviors, and things in models, like use case (types). But of course, it is very hard to speak strictly all the time. So the term "use case" is often used, instead of the proper term "use case instance," to say that a use case communicates with actors.

Ivar Jacobson thinks that he has been very clear, from the beginning, that use cases are classes: "*In the book I made it very clear that use cases were classes, that could be instantiated and that could interact with actors (users) only*" [3]. But in fact, as in this very sentence, Jacobson was not very careful in his terminology. He rarely used the term "use case instance", and he used use case to mean either a use case (class), or a use case instance. For example, when saying that "*Customer will start the use case, but Operator will also communicate with it*" [4]. Conversely, Ivar Jacobson has sometimes used the expression "a *use case description,*" instead of just saying, "use case."

For this reason (and also for other reasons that we will see), many use case practitioners have the incorrect impression that the Extend and the Include relationship ap-

ply between use case instances. As a result, they do not fully understand their semantics.

2.2. Explanations in the Wrong Domain of Discourse

UML holds that an Extend relationship is between use cases, that is, between specifications, but at the same time it says that the condition of this relationship "*must hold when the first extension point is reached for the extension to take place*" [2]. Likewise, Jacobson tends to explain the notion of extension by talking of use case instances, but calling them use cases (as he usually did in his first book): "*What happens when a course is inserted in this way is as follows. The original use case runs as usual up to the point where the new use case is to be inserted.*" [4]

It does not make sense that a use case, which is an invariable piece of specification of a system, can be extended (or not) depending on a condition that occurs during the runtime of that system. Even more so since multiple instances of that use case may run simultaneously, each yielding its own result of the extension condition.

Of course, UML provides other explanations that are more correct. For example, the semantics clause about Extend says the following:

"If the condition of the extension is true at the time the first extension point is reached during the execution of the extended use case, then all of the appropriate behavior fragments of the extending use case will also be executed. If the condition is false, the extension does not occur. ... Note that even though there are multiple use cases involved, there is just a single behavior execution." [2].

So, it is possible to a careful reader of the UML specification to correctly understand how to interpret an Extend relationship, even though the specification is adamant that the extension does occur, or does not, after the condition is evaluated, that is, at runtime.

2.3. Misleading Explanations

UML-1 holds that "*The base use case may not be dependent of the addition of the extending use case*", while UML-2 explains:

"Note, however, that the extended use case is defined independently of the extending use case and is meaningful independently of the extending use case." [2]

Both explanations give the false impression that an extended use case is wholly independent of its extending use cases. In other words, that the extensions are not necessary, or not always necessary.

In fact, Jacobson's idea was not that "*the extended use case is defined independently of the extending use case*", but rather that it has a "*course of events that is meaningful in itself.*" As Ivar Jacobson explains in [5], an extending use case may be obtained by extracting (a description of) a flow of events from an overly complex use case. One would expect both the resulting use cases to be dependent, to some extent, on each other.

An extended use case depends on its extending use cases in that those may handle exceptions that cannot be prevented. Implementing an extended use case without im-

plementing its extending use cases may result in a system that will deadlock, or that will fail to address critical user and stakeholder requirements.

2.4. Conflation between Specification and Type

UML systematically conflates the notions of specification and type, by having a single modeling element represent both a specification and its associated type. For example, the model element Class denotes both a specification of an object (a description that can be read and instantiated), and a type (a conceptual entity – UML-2 explains that "A type represents a set of values"). As every OO programmer surely knows, the specification and the type are different – every instantiation of the class is necessarily an instance of the type, but the converse is not true.

From the definition of use case in UML-2 (see Section 2.1), it is very clear that UML also conflates the notions of specification and type of a use case. One consequence is that every use case specification implies a use case type with the same name. Even an extending use case, which typically specifies several fragments of behavior, rather than one behavior whose execution yields a result, is considered to be a use case and to define a type. However, is this a desirable situation?

In his first book, Ivar Jacobson gives an example in which the base use case is called "*Returning item*," while the extending use case is called "*Item stuck*." Jacobson, who cares about how to name use cases, has applied two different naming principles to two different kinds of use cases. "*Returning item*" gives an idea of the result expected from executing the use case – this naming principle is coherent with the very definition of use case in UML. "*Item stuck*" does not denote an expected result, but rather an exception situation. It appears that Jacobson considers ordinary use cases and extending use cases to be in two separate categories, and that he would apply different typing principles to them. Since neither UML's ontology nor Jacobson are explicit about the types of use cases, simply exploring this topic with UML experts is very difficult.

Likewise, it is difficult for UML experts to explore whether there should be dependence between relationships among types (i.e., generalizations), and relationships among specifications (i.e., inheritance, Extend, and Include). UML constrains generalization and inheritance to be aligned, but without being explicit about the implications of that choice. More importantly, a debate still exists about whether there is dependence between Extend and generalization, or between Include and generalization. The only way to settle this debate would be to look at what the types for use cases exactly are, but the UML framework does not provide any help to do so, quite the contrary.

2.5. Use Case Diagrams

There is only one kind of use case diagram in UML. All the relationships and associations of use cases are declared in such diagrams. Therefore, a naive reader may not understand that there is an essential difference between an Extend relationship and an association between a use case and an actor: the association represents in fact rela-

tionships, called links, between instances of the use case and (instances of) the actor; on the other hand, the Extend relationship really is between use cases, and it does not represent relationships between instances of the use cases.

Likewise, a UML reader may not understand the essential difference between an Extend relationship, which relates specifications, and a generalization relationship, which relates types.

3. The Ontology of the RM-ODP

As noted in the UML standard, "a major purpose of modeling is to prepare generic descriptions that describe many specific items." [1]. UML goes on to explain:

"This is often known as the type-instance dichotomy. Many or most the modeling concepts in UML have this dual character, usually modeled by two paired modeling elements, one represents the generic descriptor and the other the individual items that it describes. Examples of such pairs in UML include: Class-Object, Association-Link, UseCase-UseCase Instance, Message-Stimulus, and so on" [1].

As can be noted, there is no systematic approach in the UML terminology to relate a "generic description" to its related "specific items". In that respect, the RM-ODP is much more systematic and orthogonal, as can be seen from the definition of its concept of template.

<X> Template*: The specification of the common features of a collection of <X>s in sufficient detail that an <X> can be instantiated using it. <X> can be anything that has a type. ... A template may specify parameters to be bound at instantiation time. ... Templates may be combined according to some calculus. The precise form of template combination will depend on the specification language used. [6]*

In this definition, the string "<X>" denotes a parameter for which the concept of ODP template is applicable. So, ODP speaks of an object template[1] whereas UML speaks of a class. Likewise, ODP speaks of an "operation template" rather than of a "method."

Note that the ODP terminology gives precedence to the "specific items", rather than to the "generic descriptions". So, in ODP, the "specification of the conveyance of information from one instance to another" would be called a "message template," leaving the term "message" available for denoting "the passing of information from one instance to another" ("stimulus" in UML).

3.1. Specifications vs. Types

In explaining the "type-instance dichotomy", UML gives the example of the pair "Class-Object", even though it is widely accepted in the OO community that a class is not a type. In fact, UML conflates the notions of type and specification, and correspondingly, the notions of generalization (a.k.a. supertyping) and inheritance.

[1] The ODP concept of "template" discussed here should not be confused with the homonymous concept in UML and C++. It is also different from the concept of template, or form, that analysts fill in for producing a textual specification.

To the contrary, the RM-ODP makes an essential difference between the notions of type and template. In particular, a type in ODP is considered to be a predicate rather than a specification.

***Type (of an <X>)**: Type (of an <X>): A predicate characterizing a collection of <X>s. An <X> is of the type, or satisfies the type, if the predicate holds for that <X>. A specification defines which of the terms it uses have types, i.e. are <X>s. In RM-ODP, types are needed for, at least, objects, interfaces and actions. The notion of type classifies the entities into categories, some of which may be of interest to the specifier … [6]*

The RM-ODP considers that types are predicates and ignores the fact that types, too, have specifications. Types classify entities into categories – how a type is specified to obtain that result is not important. On the other hand, specifications, not predicates, are of interest with templates[2] – they are needed (by factories, performers or implementers) for instantiating individual elements.

In some cases, as with the concept of class in UML and in most OO programming languages, a template is implicitly associated with a type. This situation is captured (and generalized to other elements than objects) in the RM-ODP with the notion of template type.

***Template type (of an <X>)**: A predicate defined in a template that holds for all the instantiations of the template and that expresses the requirements the instantiations of the template are intended to fulfil... [6]*

It is important to note that the template type is not defined to be the most specific predicate that could be derived from the template. Many implantation details are indeed deemed irrelevant to users of instantiations of the template. For example, the details of a method are not deemed pertinent for typing an object.

Instances vs. Instantiations. While UML uses "instance" and "instantiation" interchangeably, the RM-ODP makes a useful distinction between these two concepts.

Instantiation (of an <X> template): An <X> produced from a given <X> template and other necessary information. This <X> exhibits the features specified in the <X> template… [6]

Instance (of a type): An <X> that satisfies the type. [6]

While an individual element is an instantiation of at most one template, it may be an instance of many types (including template types).

Relationships between Types and between Specifications. The only relevant relationships between types are those of subtype/supertype.

***Subtype/supertype**: A type A is a subtype of a type B, and B is a supertype of A, if every <X> which satisfies A also satisfies B. [6]*

To the contrary, templates are specifications, and they have a very different kind of relationship between them – they can be combined or derived from one another using some calculus. For example, inheritance is a kind of derivation relationship between templates.

The RM-ODP makes a clear distinction between generalization and inheritance:

---

[2] "Templates are something that you can contemplate [i.e., see]", said Elie Najm, a French contributor to the RM-ODP.

"The inheritance hierarchy (where arcs denote the derived class relation) and the type hierarchy (where arcs denote the subtype or subclass relation) are therefore logically distinct, though they may coincide in whole or in part." [6]

4. Use Case Concepts from an RM-ODP Perspective

From an RM-ODP perspective, the fundamental concept would be that of a use case individual, that would simply be called a *use case*. Its definition, inspired by that of use case in UML-2, could be as follows.

Use case: a set of actions performed by a system, which yields an observable result that is, typically, of value for one or more actors or other stakeholders of the system.

As we will now see, the related concepts are easily derived from the generic ODP definitions.

4.1 Templates and Specifications

A **use case template**, following the OPD terminology[3], is the specification of the common features of a collection of use cases, in sufficient detail that a use case can be instantiated (by a programmer) using it.

A use case template is the *full descriptor* of a use case. This full descriptor is obtained from a base use case specification by applying inheritance to it, as well as all its inclusions and extensions. Depending on context, a use case name, e.g. "Returning item" represents a partial specification (a use case as it is written by analysts), a use case template (the specification implemented by programmers), or a type of use case (see next section).

What UML calls an extending use case is in ODP a partial specification of a use case (the specification of some of the common features of a collection of use cases). The name of an extending use case, e.g. "Item stuck", can denote a partial specification of a use case, and perhaps a type (see next section). It cannot denote a use case template since an extending use case may extend several base use cases.

The question of whether an included use case specification may induce a use case template is more complicated. The answer is clearly positive if the use case specification, or more precisely the full descriptor that it entails, may be instantiated on its own. It is clearly negative if the "included use case" provides no observable result to an actor or to a stakeholder of the system.

That latter case calls for the introduction of a new concept that would be a generalization of the concept of a use case: First, an **occurent** is defined as anything that happens, as opposed to something that persists [7]. Second, a **system action** is defined as a an occurent in which the efficient cause of the occurent is the system [8]. But the condition of the provision of providing an entirely observable result is not yet included. A use case is then the externally observable specialization of system action.

[3] The ODP concept of "template" discussed here should not be confused with the homonymous concept in UML and C++. It is also different from the concept of template, or form, that analysts fill in for producing a textual specification.

The full descriptors of "included use cases" would then be "**system action templates**", some of which would be use case templates as well.

4.2 Types of Use Cases

We do not intend in this paper to fully define the way use cases shall be typed. We leave the problem to the designers of use case methods. However, we can explain the ODP view on this matter to them.

Two questions are to be answered: 1) In a use case model, what are the types that are defined? and 2) How are these types defined?

As a partial answer to the first question, ODP suggests that there is one type, the template type, for each use case template in the model. These types are named by the name of their template, i.e., by the name of their base use case specification.

The ODP definition of a template type (see Section 3.1) gives us an indication as to how these types shall be defined. Moreover, we can take into account the specificities of use case modeling for determining what should be meant by "the requirements the instantiations of the template are intended to fulfill." We obtain the following definition:

Template type of a use case: a predicate defined in a use case template that holds for all the instantiations of the template and that expresses the observable result that the instantiations of the template are intended to provide.

Concretely, a methodologist could therefore declare that the type of a use case is defined by the contents of the field named "Goal" in the base use case specification. A methodologist, following UML, may also decide that additional properties of the use case should also go into the making of its template type, such as its associations to actors.

If an included use case specification is the basis of a use case template, then it yields a type of a use case (as every template does). In general, this template type is unrelated (by generalization) to the template types of the including use cases.

Since an extending use case specification is not the basis of any use case template, it yields no template type. This leaves methodologists with a choice as to whether or not an extending use case specification yields a use case type. Our own inclination is to answer this question by the negative (we are comforted in that direction by the different naming discipline that Jacobson was applying to extending use case specifications, as compared to ordinary use cases, see Section 2.4). However, we can conceive that an extending use case specification (or more precisely, a clause in this specification) goes into the making up of template types.

Since the generalization relationship in UML denotes both inheritance between specifications and sub/supertyping relations between types, it is logical that "generalized use cases" also yield types.

The generalization relation between use case types needs not to be explained as it is implied by the ODP definitions of subtype and supertype. It is of course distinguished from inheritance (a reuse relation between specifications). However, the characteristics used for typing and the inheritance rules can be such that inheritance and generalization coincide.

4.3 The Include and Extend Relationships

The ODP view is that Include and Extend are relationships between use case specifications. Both mean that the two specifications they join must be combined, unconditionally, to yield a new specification.

Of course, Extend has a condition associated to it. In fact, this condition pertains to how to achieve the requested combination – that is, it is to be considered when considering the semantics of the full descriptor of the use case.

In his book "The Object Advantage," Ivar Jacobson, with his coauthors, confirms that the Include and Extend relationships apply unconditionally: "*In other words, we have defined two relations between use cases, both of which are of static character*" [5].

In his earlier book "Object-Oriented Software Engineering", Ivar Jacobson explained that Uses (now Include) may be considered as a kind of inheritance relationship [4]. Indeed, thinking exclusively in terms of specifications, an included use case contributes to a use case template in very much the same way that a generalized use case does. And so does an extending use case. However, such inheritance relationships have no implications as to the possible relationships between types.

When to Use Extend. Ivar Jacobson speaks also in terms of specifications when he explains why he has invented the Extend relationship.

"*A use case description can be rather difficult to overview if it contains too many alternative, optional or exceptional flows of events that are performed only if certain conditions are met as the use-case instance is carried out. A way of making the description 'cleaner' is to extract some of these sub-flows and let them form a use case of their own. This new use case is then said to extend the old one, if the required conditions are met. Such a construction can be achieved by using the extends association between the use cases.*" [5]

In fact, Ivar Jacobson explains two notions, which should be distinguished: *extending a behavior*, and the *Extend relationship*. Many use case practitioners, and in particular Alistair Cockburn, barely use the Extend relationship at all. Yet, when writing a use case, they often use one or several clauses describing alternative flows of events. Alistair Cockburn calls such clauses "*extensions*" ("We say extension as opposed to failure or exception so that we can include alternative success as well as failure conditions." [9]) Even though Cockburn has extensions being part of his use case, he speaks of the "*extension conditions*", and he correctly explains that these are "*the conditions under which the system takes a different behavior.*" [4]

In the above explanation, Ivar Jacobson seems to suggest that the Extend relationship must be used whenever one wants to make a use case cleaner. But his position ignores widely used techniques for writing a use case, as the specification of a normal flow of events, plus the specifications of alternative flows of events. This leaves his readers still perplex about when to use Extend. This leads us to propose a simpler, more understandable, explanation.

---

[4] Alistair Cockburn uses here the term "behavior" with its English meaning (what the system actually does) rather than with its UML meaning (a specification of what the system shall do).

The Extend relationship embodies a reuse technique for specifications of alternative flows of events. When a writer of use case finds herself in a situation where she has to write again and again the same specification of an alternate flow of events, she has the option to put this piece of specification in a so-called extending use case, and to relate this use case via the Extend relationship to all the other use cases to which it applies.

5. Summary and Conclusions

While it is possible for a careful reader of Jacobson and of the UML standards to obtain a correct understanding of the semantics of use case models, it is rather difficult to do so. Its ontology being unnatural (at odds with English), the UML standard contains numerous sentences that confuse the picture between use cases, use case instances, and use case types. It is not surprising then that many use case practitioners believe that Extend and Include are relationships between use case instances.

The ontology of the RM-ODP, on the other hand is more natural and more easily applicable to other contexts than OO modeling[5]. It is indeed easier to be rigorous in saying use case type or use case template whenever this is what is meant, than to always add "instance" after "use case" in the other situations. Applying the same rigor, one would explain Include and Extend for what they are, that is relationships between specifications. By separating the relations of inheritance between specifications, and generalization between types, we see that we can reconcile the positions of Jacobson who explained Uses and Extends in terms of inheritance [4], and Anthony Simons who provided in [10] compelling arguments why Extend could not be mistaken for inheritance[6].

Following the work that was done in the foundations of logic in the 20th century [11], the RM-ODP ontology gives precedence to the most fundamental concept, that of the individual use case, which can be defined for what it is. Immediately, it becomes clear that an instance of an "included use case" does not necessarily fit that definition of a use case. The inescapable conclusion, so far unseen by UML, is that another concept (say system action) is needed to explain use case modeling.

This paper does not propose final answers to questions about UML semantics. Rather, we hope to have provided UML experts with new tools to analyze the problems from a new perspective, to find appropriate solutions, and to communicate more effectively about them.

---

[5] OO programmers are of course used to the terminology of object and class.

[6] Anthony Simons was clear that he was thinking of the inheritance relationship as specialization, which is a conflation of a specification and a type relationships.

6. Acknowledgements

We acknowledge the contributions of Joaquin Miller to the years of discussions that lead to this paper. We are also grateful to an anonymous reviewer for his or her valuable comments and suggestions.

7. Bibliography

1. OMG, OMG Unified Modeling Language Specification v. 1.5. 2003, OMG.
2. OMG, UML 2.0 Superstructure Specification (draft recommendation in the finalization phase). 2003, OMG.
3. Fowler, M., et al. Question Time! about Use Cases. in 13th ACM Conference on Object-Oriented Programming, Systems, Languages and Applications-OOPSLA'98. 1998. Vancouver, B.C.: ACM SIGPLAN Notices.
4. Jacobson, I., et al., Object-Oriented Software Engineering--A Use Case Driven Approach. 1992, Reading, Massachusetts: Addison-Wesley, 1992. 524.
5. Jacobson, I., M. Ericsson, and A. Jacobson, The Object Advantage: Business Process Reengineering with Object Technology. ACM Press Books. 1995: Addison-Wesley. 347 pp.
6. ISO/IEC and ITU-T, Open Distributed Processing - Basic Reference Model - Part 2: Foundations, in Standard 10746-2, Recommendation X.902. 1995.
7. Sowa, J.F., Knowledge Representation: Logical, Philosophical, and Computational Foundations. 1999: Brooks/Cole Pub Co.
8. Barnes, J., Aristotle's Posterior Analytics. 2nd ed 1994 ed. 1975, Oxford: Claredon Press.
9. Cockburn, A., Writing Effective Use Cases. Object Technology Series. 2000.
10. Simons, A.J.H. Use Cases Considered Harmful. in 29th Conf. Tech. Obj.-Oriented Prog. Lang. and Sys., (TOOLS-29 Europe). 1999: IEEE Computer Society.
11. Strawson, P.H., Introduction to Logical Theory. 1963: Routledge Kegan & Paul.