

Digging into Use Case Relationships

Gonzalo Génova, Juan Llorens, Víctor Quintana

Computer Science Department, Carlos III University of Madrid,
Avda. Universidad 30, 28911 Leganés (Madrid), Spain
{ggenova, llorens}@inf.uc3m.es, victorq@ie.inf.uc3m.es
<http://www.inf.uc3m.es/>

Abstract. Use case diagrams are one of the key concepts in the Unified Modeling Language, but their semantics and notation have some gaps that lead to frequent misunderstandings among practitioners, even about very basic questions. In this paper we address some issues regarding the relationships in which use cases may take part. The Include and Extend relationships between two use cases have presently an inconsistent definition, since they are represented as stereotyped dependencies, but they are not true dependencies in the metamodel. Besides, the direction of the dependency arrow in the Extend relationship can be misleading, unnatural and difficult to understand for the common practitioner. Finally, we show also some conceptual problems regarding the included or extending use cases, which in our opinion are not true use cases.

Introduction

There are two kinds of relationships in which use cases may take part in a use case diagram: relationships between two use cases, and relationships between a use case and an actor¹. They are always binary relationships.

There is something strange about relationships between two use cases, which are *directed relationships* from a source use case to a target use case. In the first versions of UML, up to version 1.1, they were modeled as stereotyped generalizations, named "uses" and "extends", following Jacobson's original ideas [5]. After adoption by the OMG (Object Management Group) UML went through two revisions, and the old stereotyped «uses» and «extends» generalizations were quietly replaced in version 1.3 by the new stereotyped «include» and «extend» dependencies², which have survived without change in version 1.4. Both the names and the semantics of the relationships were deliberately changed by the UML authors, but the whys and the hows were never widely explained to the general public [13, 15].

In fact, this change has introduced a conflict between the notation and the metamodel. The new relationships are graphically represented as stereotyped dependencies but, as we will show, they are *not true dependencies* in the metamodel,

¹ A previous version of this paper contained a Section on the relationships between use cases and actors, but it had to be suppressed due to space problems.

² Whether they are dependencies, or not, is one of the main subjects of this paper.

but direct subclasses of the Relationship metaclass. This is in itself a source of confusion about the true nature of these relationships. What kind of relationships are really «include» and «extend»? Should the metamodel or the notation prevail, or neither?

There are other common misunderstandings about use case models that are induced by a lack of definition and a misleading notation in use case relationships. Even though some authors suggest that use case relationships should be confirmed as usage-dependencies [7], we think that dependency in itself is not a clear concept, used in UML rather as a supertype for everything that is neither generalization nor association [UML 2-33]. Practitioners very often complain about the perplexing direction of the «extend» dependency. Besides, we have observed a general tendency towards misinterpreting use case dependencies as control flow relationships between processes, leading to a confusion between use case diagrams and activity diagrams. Our rather drastic proposal is to avoid at all the decomposition of use cases by means of use case relationships, allowing only self-contained use cases that are directly connected with actors.

The remainder of this paper is organized as follows. Section 2 explains in detail the conflict between notation and metamodel for use case relationships in the current version of UML. Section 3 examines the directionality of use case relationships, exposing some conceptual problems about the nature of these relationships. Finally, Section 4 proceeds by showing the risks of use case relationships for a correct understanding of a use case model.

Since this is a conceptual research about the nature of use case relationships and their official definition, our main source has been The OMG Unified Modeling Language Specification [9], more briefly referred to as "The Standard". This document is properly the only one which is truly "official", but there are many semantic questions that are poorly explained in it, so that we have turned to the works of the original authors of the UML in search for a clarification: The Unified Modeling Language Reference Manual [12], which seemed an obvious choice, and The Unified Modeling Language User Guide [2]³. On the other side, we cite the User Guide not because we consider it a particularly reliable source, but because it is probably the main source for many modelers, so that we think it is important to show its virtues and deficiencies. We quote version 1.4 of the Standard, and we have checked that there have been no significant changes from version 1.3 regarding these topics.

³ In the remaining of this paper, these three references will be cited as "UML" for the UML Specification, "RM" for the UML Reference Manual, and "UG" for the UML User Guide, followed by page numbers.

1. A misleading notation for use case relationships

1.1. In UML version 1.1

In UML (up to version 1.1 [10, 11], before adoption by the OMG) there were two kinds of relationships between use cases: "extends" and "uses". They were both represented by generalization arrows, respectively labeled with the stereotypes «extends» or «uses» (see Figure 1).

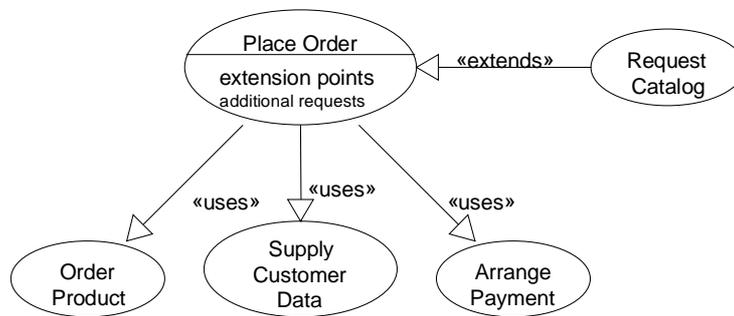


Figure 1. An example of use case relationships as stereotyped generalizations, extracted from UML Notation v1.1 [11] (p. 79)

The metamodel for use cases (see Figure 2) was not very developed in version 1.1, so that these two relationships had not any prominent place in it: there didn't exist any special metaclasses for them. Therefore, we must conclude that they were true stereotyped generalizations, available to the UseCase metaclass because it is a subclass of Classifier, which is a subclass of GeneralizableElement. The third well-formedness rule for use cases imposed that these were the only relationships allowed between use cases [10] (p. 92).

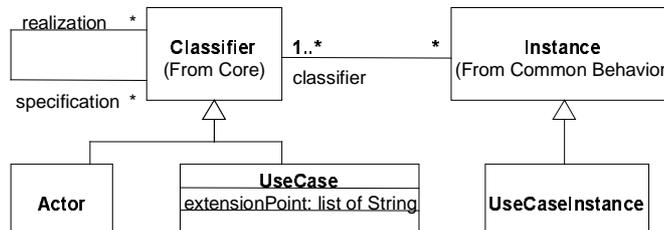


Figure 2. Metamodel of use cases, extracted from UML Semantics v1.1 [10] (p. 90)

1.2. In UML version 1.4

Because of the conceptual problems caused by the definition of relationships between use cases as generalizations, and the fact that developers completely disregarded the generalization semantics in the way they employed them [13, 14], UML later abandoned this approach in version 1.3 in favor of a more plausible definition, based on a notion of dependency which has survived without change in version 1.4. Nevertheless, this definition has not been made at all clear in the current version, as we are going to show: use case relationships are *shown* as dependencies, but they are *not true* dependencies.

The former «uses» and «extends» *stereotyped generalizations* have supposedly evolved into «include» and «extend» *stereotyped dependencies*, whereas a new plain generalization is also allowed. In Figure 3 we can see the equivalent of Figure 1 in the new notation. Solid lines with hollow arrowheads (generalization) have been substituted by dashed lines with open arrowheads (dependency).

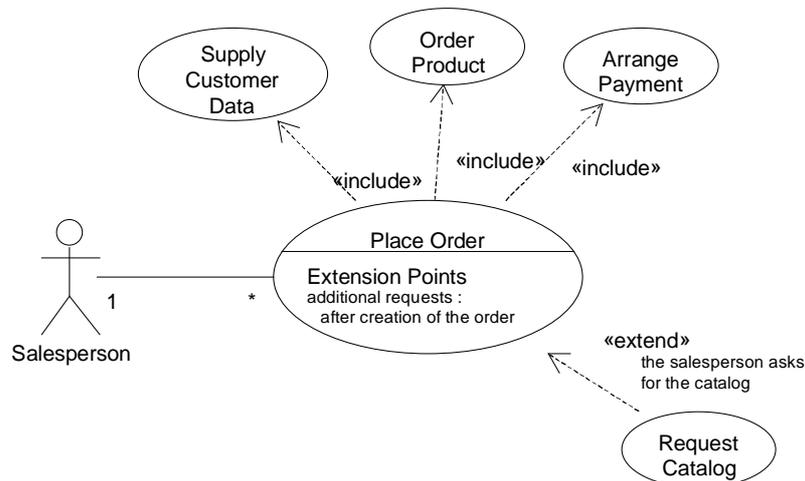


Figure 3. An example of use case relationships as stereotyped dependencies, extracted from UML Specification v1.4, p. 3-99. Compare with Figure 1. We have respected the original layouts in both diagrams, even though they are upside down and this could mislead the eye of the reader. Notice the differences in the descriptions of extensions and in the addition of an actor to the diagram.

Indeed, this notation states that we are dealing now with some kind of dependency, as we were dealing before with some kind of inheritance. The User Guide defines these use case relationships as stereotypes of dependencies [UG 227, 228]. The Reference Manual also says that "an extend relationship is a kind of dependency" [RM 490], that an include relationship "is a dependency, not a generalization" [RM 491], and that "an extend relationship or an include relationship is shown by a dependency arrow" [RM 493]. Everything seems clear. What is the problem, then?

Let's have a closer look at the definitions in the Standard. Remarkably, the description of the notation for these two relationships carefully avoids either the word "dependency" or defining them as dependencies [UML 3-98]:

- "An extend relationship between use cases is *shown by a dashed arrow with an open arrow-head* from the use case providing the extension to the base use case. The arrow is labeled with the keyword «extend»."
- "An include relationship between use cases is *shown by a dashed arrow with an open arrow-head* from the base use case to the included use case. The arrow is labeled with the keyword «include»."

A similar caution is not observed when describing the other possible relationship between use cases plainly as a generalization [UML 3-98].

These remarks could be disdained as non-significant subtleties, and a pragmatic position could be assumed: "they are represented as dependencies, so they are dependencies", but the point is that the metamodel itself does not define the use case relationships as dependencies. In fact, the metaclasses Extend and Include are direct subtypes of the metaclass Relationship (see Figure 4 and compare with Figure 2).

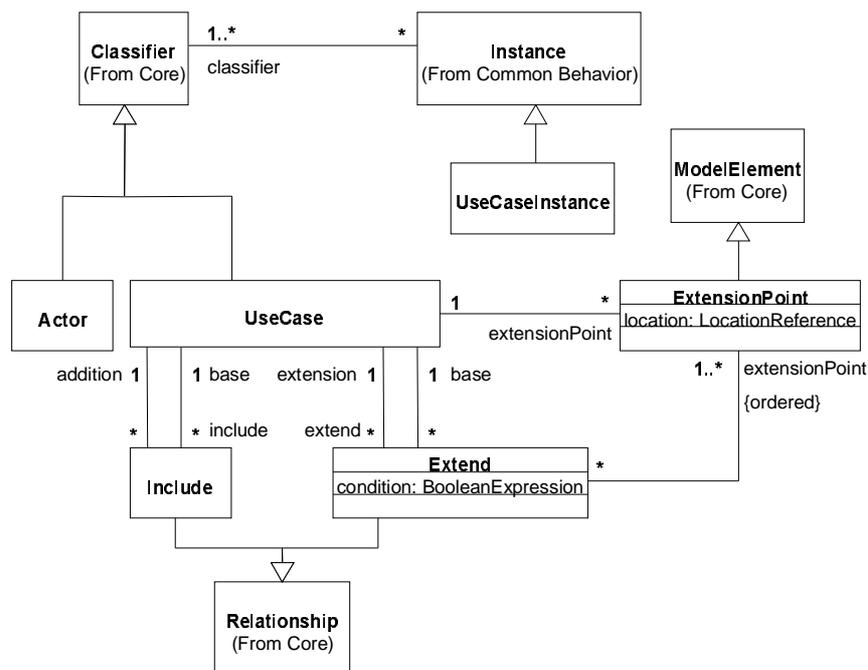


Figure 4. Metamodel of use cases, extracted from UML Specification v1.4, p. 2-135.

The Relationship metaclass did not yet exist in v1.1, it was introduced later as a supertype of all kinds of relationships (Flow, Generalization, Association, Dependency, etc.). The Dependency metaclass has its own subtypes; Include and Extend are not subtypes of Dependency, but direct subtypes of Relationship (see Figure 5).

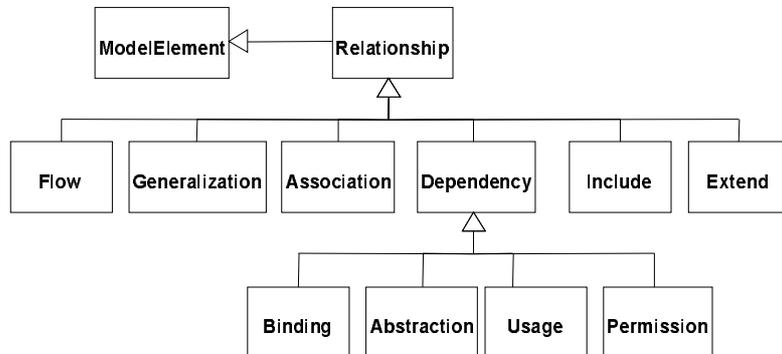


Figure 5. The Relationship metaclass and its subtypes, extracted from UML Specification v1.4, pp. 2-14, 2-15, 2-135.

Aside from the graphical presentation of the metamodel, the Standard does not say that Include and Extend are dependencies, neither in the Abstract Syntax section [UML 2-136, 2-137], nor in the Detailed Semantics section [UML 2-143, 2-144], and the list of kinds of Dependency does not include them [UML 2-33, 3-91]. A stereotype is used in UML to extend the language with new modeling elements, which are derived from existing ones [UML 2-78] [8]. Predefined stereotypes are equivalent to subclasses in the metamodel [16] that inherit, among others, the notation of the parent class. For example, a «use» dependency in a model is possible because Usage is a subclass of Dependency in the metamodel, and it is represented with the same notation, a dashed arrow with open arrowhead, adding the stereotype's keyword. In the case of Include and Extend, they are direct subtypes of Relationship, but they use the notation of Dependency instead, creating a conflict.

From this scrutiny we can conclude that:

- According to the Standard, Include and Extend are intentionally not dependencies, but relationships on their own (direct subtypes of Relationship).
- The Standard is contradicted by the User Guide and the Reference Manual, where Include and Extend are defined as stereotyped dependencies.
- The Standard proposes a misleading notation, since it represents Include and Extend graphically as stereotyped dependencies; this is in itself a source of confusion.

This conflict could be easily solved by changing the metamodel and making Include and Extend subtypes of Dependency. But, before taking this step, the true nature and convenience of these relationships must be enlightened. Besides, if Include and Extend are finally not dependencies, then the direction of the arrow should be established with a criterion that is not that of a dependency.

2. A misleading direction of dependency in the Extend relationship

Include and Extend are said to be *directed relationships* [UML 2-136, 2-137], which is conveniently represented by an arrowhead on the corresponding end of the relationship. "The include relationship points at the use case to be included; the extend relationship points at the use case to be extended" [RM 66]. In other words, UML tells us to draw the arrow *from the base to the inclusion* (or addition, in terms of the metamodel, see Figure 4), and *from the extension to the base* (see Figure 7). Thus, in the «extend» relationship, the base of the relationship is not the base of the arrow.

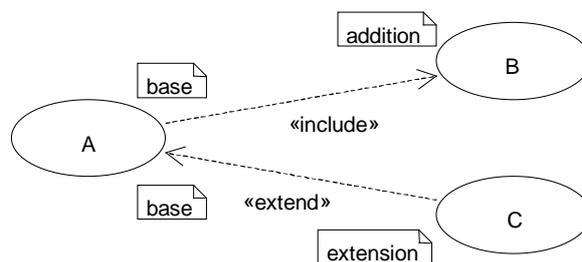


Figure 7. Include and Extend relationship arrows confronted with the underlying metamodel

Perhaps not surprisingly, most people think intuitively that the «extend» arrow should go the same way round as the «include» arrow [17], that is, both arrows should point from the base case towards the inclusion or the extension. This is probably due to a poor understanding of the difference between these relationships by practitioners, a problem which has been denounced long ago, and about which even the experts disagree [4]. This difficulty is not completely solved by assuming that Include means *reuse* while Extend means *insertion*, or *alternative*, or *exception*, since these concepts are not that clearly independent: if B is reused by A, then in some way B is being inserted into A; if C is inserted into A, or is an alternative to A, or an exception, then in some way C is being reused by A (or, at least, "used"). We understand practitioners when they find difficulties in deciding whether they should use Include or Extend in their models.

The difference between Include and Extend is understood by some authors as *conditional versus unconditional insertion*: a use case is inserted into another use case, in analogy with subroutine call semantics [1, 7]. In this way, the inserted use case is easily seen as an independent reusable, encapsulated module, while the base use case in which the insertion is made is seen as dependent on the insertion. With this notion in mind, practitioners are perplexed when they are told that they must draw the «extend» arrow in reverse direction: they do not see a true dependency in it. As a rule of thumb, one can think that the arrow goes from the "grammatical subject" to the "grammatical object" of the corresponding *verb* "include" or "extend". For example, in Figure 3 the Request Catalog use case *extends* the Place Order use case, which in turn *includes* the Order Product use case. But this "grammatical rule" that helps in drawing the arrows does not seem adequate to understand the true underlying dependencies between use cases.

In fact, the same authors mentioned above [1, 7] think that there is not any fundamental difference between the Include and Extend relationships: they would be semantically equivalent. Extend would be considered "if (condition) then ...", whereas Include would be considered "if (TRUE) then ..." [7]; this would be consistent with the original idea of an extension being a guarded block of functionality that could be inserted into a main use case upon fulfillment of a certain condition [13], although Jacobson viewed the extensions as interrupts rather than branches [5]. If they were truly equivalent, both relationships could be collapsed into a single relationship analogous to what the OPEN/OML approach calls the "invokes" relationship [3], and of course they should be drawn in the same direction.

In order to establish the rightness of this approach, the true dependencies underlying Include and Extend must be exposed, which will lead to a better understanding of these relationships, their similarities, and their differences.

- **Include.** According to the Reference Manual, "the base use case can see the inclusion and can depend on the effects of performing the inclusion, but neither the base nor the inclusion may access each other's attributes" [RM 297]. This indicates a dependency from the base to the inclusion, as the arrow accordingly states⁴.
- **Extend.** On the other side, "an extension use case in an extend relationship may access or modify attributes defined by the base use case. The base use case, however, cannot see the extensions and may not access their attributes or operations. The base use case defines a modular framework into which extensions can be added, but the base does not have visibility of the extensions" [RM 273]. That is, the base use case defines the extension points in which it would accept extensions, but it knows nothing more about these extensions. This indicates a dependency from the extension to the base, as the UML authors advocate.

In consequence with these definitions, the dependencies seem to be rightly expressed in UML: the base use case sees the inclusion and is seen by the extension. We can compare Include to a *go-to* instruction (Place Order performs a *go-to* Order Product), whereas Extend is to be compared rather with a *come-from* instruction (Request Catalog performs a *come-from* Place Order). Even if we accepted that the dependencies are right, the whole scheme is rather unnatural and difficult to understand for the common practitioner.

Nevertheless, in the Extend dependency the base use case must define the extension points, which makes it not completely independent of the extending use cases. In addition, other authors [13, 14] think that making the variant or exceptional behavior depend on the normal behavior is not a *logical dependency*, but an artifact of the order followed in the analysis procedure (first write the normal cases, then write the extensions on separate sheets and state where in the normal cases they should be

⁴ Unfortunately, some lines below in the same page, we read: "The inclusion use case may access attributes or operations of the base use case", which contradicts the previous statement and would induce a dependency from the inclusion to the base. We think this is a weird error in the text, since it prevents the desired encapsulation and reuse semantics, as stated right after: "The inclusion represents encapsulated behavior that potentially can be reused in multiple base use cases".

inserted): this is like saying that all branches of a multi-branch statement depend logically on one distinguished branch. In this view the «extend» arrow points in the wrong direction (from the extensions to the base) but the right direction would not be the reverse one (from the base to the extensions). Instead, a radically different scheme must be followed: from a superordinate selection node to the base and all the extensions equally. Unfortunately, this scheme is adequate for modeling *alternatives*, but it is less adequate for *insertions*, and it is not adequate for *exceptions*.

Strangely, the current version of the metamodel does not provide an InclusionPoint metaclass for the Include relationship, to represent the place where the inclusion takes place, equivalent to the ExtensionPoint for the Extend relationship (see Figure 4), even though this is a natural concept and it is even mentioned in the Reference Manual [RM 297, 491]. There is another more fundamental difference between the inclusion and the extension that prevents a true equivalence among them. In the «include» relationship, the included use case is inserted at one location in the base use case and represents encapsulated behavior [UML 2-143]; it follows a *subroutine call semantics* [6]. Conversely, in an «extend» relationship, upon fulfillment of a single condition, the different fragments of behavior defined in the extension are inserted simultaneously and coordinately at different locations in the base use case [UML 2-144]; therefore, it follows *interleaving semantics* [6], and consequently the extension does not define any encapsulated behavior. The problems of interleaving semantics have already been exposed [6]: interleaving violates encapsulation and other fundamental principles of object-orientation, a fault that should be more than enough to discard interleaving and promote subroutine call semantics (that is, encapsulation) also for extend relationships. In the kingdom of encapsulation, interleaving semantics is not easy to understand.

3. The risks of use case relationships

A use case execution is commonly considered a use case instance [5, 6, 13, 14], that is, "the performance of a sequence of actions specified in a use case" [UML 2-138]. In earlier works on use cases [5], inclusions and extensions were called "abstract" use cases, because they were not instantiable (that is, they were not full and independent external software services), in contrast with "concrete" use cases, which were directly connected with actors and represented a complete service offered to an actor. That is, when an actor requires a service from a concrete use case that has an abstract use case included in it or extending it, there is only one running instance, that of the concrete use case. The concrete use case completes its description with that of the included or extending abstract use case; this "usage dependency" exists only at the classifier level, there isn't any relationship between use case instances. The term "abstract use case" is reminiscent of the time in which use case relationships were stereotyped generalizations, and it has been abandoned in the current version of UML⁵. Therefore, in order to avoid this term, which is particularly misleading, we are

⁵ That is, the term "abstract use case" does not mean any more an included or extending use case. Of course, since use cases are classifiers, there may exist "abstract" use cases in the

going to use the terms "primary use case" and "secondary use case". *Primary use cases* are directly connected to actors, whereas *secondary use cases* are inclusions and extensions.

In this section we are going to challenge the notion of "use case relationship". The noble intention behind inclusions and extensions is removing redundancy from the documentation: "Organizing your use cases by extracting common behavior (through include relationships) and distinguishing variants (through extend relationships) is an important part of creating a simple, balanced, and understandable set of use cases for your system" [UG 228]. However, this intention fails due to several reasons.

First of all, *secondary use cases are not true use cases*. Let's have a look at some definitions. A use case is "a description of a set of sequences of actions, including variants, that a system performs to yield *an observable result* of value to an actor" [UG 222], "the specification of sequences of actions, including variant sequences and error sequences, that a system, subsystem, or class can perform *by interacting with outside actors*" [RM 488]. "Each use case specifies a service the entity provides to its users; that is, a specific way of using the entity. The service, which is initiated by a user, is *a complete sequence*" [UML 2-141]. "A pragmatic rule of use when defining use cases is that each use case should yield some kind of observable result of value to (at least) one of its actors. This ensures that the use cases are complete specifications and *not just fragments*" [UML 2-145].

Let's see why these definitions are not satisfied by "secondary use cases":

- **Included use case.** Its purpose is to factor out and *reuse common behavior* among use cases. As a description of encapsulated behavior [UML 2-143], it could be designed to be a complete sequence that yields an observable result to the actor. However, it is more probable that it is an incomplete sequence that yields only a *partial result*, more easily reused by several primary use cases: "A use case can simply incorporate the behavior of other use cases as fragments of its own behavior. This is called an include relationship" [RM 66]. It is not that easy that a *fragment of behavior* can be at the same time a complete behavior. Typically, an included use case is *not directly connected to an actor*, but only indirectly through a primary use case⁶. An included use case does not interact with the actor, it does not specify a full service (a complete sequence of interactions), and it does not yield an observable result, therefore it is not a true use case.
- **Extending use case.** Its purpose is to *distinguish variant behavior* from the primary use case. Originally, the extend relationship was intended to represent *insertions as well as alternatives and exceptions*, but it has been

general sense applied to any classifier, but these abstract use cases must not be confounded with included or extending use cases. A true abstract use case would be indirectly instantiable via its subclasses, but this is not the pretended meaning for inclusions or extensions.

⁶ However, the Reference Manual contains an example [RM 27] of an included use case "make charges" that is simultaneously connected to an actor "credit card service", that is, a use case that is both primary and secondary, being therefore instantiable and non-instantiable at the same time. We doubt that this example is realistic. We think rather that this actor should be connected to a different primary use case, which would include the "make charges" secondary use case.

shown that insertion semantics is inadequate to model exceptions and alternatives [13, 14]. In addition to the arguments given for the included use case, which are valid here too, an extending use case may contain different fragments of behavior to be inserted into different locations (interleaving semantics), what makes it even harder to consider the extending use case as an independent external service.

Summing up, secondary use cases are not true use cases simply because they do not specify a service that a system gives to an actor. If, having observed "commonalities between use cases" [UML 2-143] or variant behaviors, we want to organize and simplify the set of use cases, a different concept must be defined in the metamodel, such as "common functionality" or "variant functionality"; otherwise, the definition of "use case" as "a coherent unit of functionality" [UML 3-96] must be changed to admit services or functions that are incomplete or that are not directly related to the actors.

Finally, the representation of secondary use cases in a model, or "common functionalities", has other, maybe worse, risks. In our opinion, the use of dependency-like use case relationships and, consequently, use cases that are not complete and self-contained, convey *a dangerous twist towards the expression of sequential processing*, or flow of control, in a use case diagram. We think that the goal of use cases is to express self-contained functionality, without expressing any sequence between those functionalities. Otherwise, these functionalities become processes that invoke one another. We have observed this phenomenon in many practitioners: they fail to understand a use case diagram because they see rather an activity diagram, or a process decomposition diagram.

Conclusions

In our experience in using and teaching UML we have learned that, against what it would seem at first sight, it is not that easy to represent a system's functionality by means of use case diagrams. There are some very basic questions about the meaning of the symbols in the diagram that do not have an easy answer, such as what is the meaning of a use case that is not directly connected to an actor, what is the meaning of the arrows between use cases, why is the arrow for the Extend relationship "in the reverse" direction, and so on.

After the last revisions of UML (currently 1.4), use case relationships have been put in an inconsistent state: they have the notation of stereotyped dependencies, but the semantics defined in the metamodel is not that of a dependency. It seems that a simple change in the metamodel would amend this contradiction, but before taking this step the true nature and convenience of use case relationships must be examined.

The direction of the dependency in the Extend relationship is misleading. Even if it were shown to be correct, it is nevertheless unnatural and difficult to understand for the common practitioner. Some authors propose a fundamental equivalence between Extend and Include, understood as conditional versus unconditional insertion. But this cannot be true in the present state of the language, since Extend's direction is opposite to Include's, and more fundamentally because of the interleaving semantics of the

Extend relationship. An InclusionPoint metaclass should be added to the metamodel to complete the equivalence, too.

Moreover, we have challenged the utility of these relationships. We have shown that included or extending use cases are not true use cases, because they are not complete and coherent units of functionality that yield an observable result to the actor. Besides, there is a danger to take the intended dependencies for control-flow relationships, leading to the confusion between use case diagrams and activity diagrams.

Acknowledgements

The authors would like to give thanks to Anthony Simons and especially to Pierre Metz for the interesting personal communications that have served as food for thought in conceiving and writing this paper. The anonymous reviewers have also provided valuable ideas and suggestions.

References

1. Klaas van der Berg, Anthony J.H. Simons. "Control-Flow Semantics of Use Cases in UML". *Information and Software Technology*, 41(10):651-659, July 1999.
2. Grady Booch, James Rumbaugh, Ivar Jacobson. *The Unified Modeling Language User Guide*. Addison-Wesley, 1998.
3. Donald Firesmith, Brian Henderson-Sellers, Ian Graham. *The OPEN Modeling Language (OML) Reference Manual*. Cambridge University Press, 1998.
4. Martin Fowler, Alistair Cockburn, Ivar Jacobson, Bruce Anderson, Ian Graham. "Question Time! About Use Cases", *Proceeding of the 13th ACM Conference on Object-Oriented Programming, Systems, Languages and Applications-OOPSLA'98*, October 18-22, 1998, Vancouver, British Columbia, Canada. ACM SIGPLAN Notices, 33(10):226-229.
5. Ivar Jacobson, M. Christerson, P. Jonsson, G. Övergaard, *Object-Oriented Software Engineering: a Use Case Driven Approach*, Addison Wesley, 1992.
6. Pierre Metz. "Against Use Case Interleaving", *The Fourth International Conference on the Unified Modeling Language-UML2001*, October 1-5, 2001, Toronto, Ontario, Canada. Springer Verlag, Lecture Notes in Computer Science 2185, pp. 472-486.
7. Pierre Metz. Personal communications to the authors, October 28th and December 16th, 2001.
8. Joaquin Miller. Post to the Precise UML Group Mailing List (<http://www.cs.york.ac.uk/puml/>), January 12th, 2002.
9. Object Management Group. *Unified Modeling Language Specification*, Version 1.4, September 2001 (Version 1.3, June 1999).
10. Rational Software Corporation, *Unified Modeling Language Semantics*, Version 1.1, September 1997.
11. Rational Software Corporation, *Unified Modeling Language Notation Guide*, Version 1.1, September 1997.
12. James Rumbaugh, Ivar Jacobson, Grady Booch. *The Unified Modeling Language Reference Manual*. Addison-Wesley, 1998.

13. Anthony J.H. Simons. "Use cases considered harmful", *Proceedings of the 29th Conference on Technology of Object-Oriented Languages and Systems-TOOLS Europe '99*, June 7-10, 1999, Nancy, France. IEEE Computer Society Press, 1999, pp. 194-203.
14. Anthony J.H. Simons, Ian Graham: "30 Things that go wrong in object modelling with UML 1.3", chapter 17 in Kilov, H., Rumpe, B., Simmonds, I. (eds.): *Behavioral Specifications of Businesses and Systems*. Kluwer Academic Publishers, 1999, 237-257.
15. Anthony J.H. Simons. Personal communication to the authors, December 5th, 2001.
16. Anthony J.H. Simons. Post to the Precise UML Group Mailing List (<http://www.cs.york.ac.uk/puml/>), January 31st, 2002.
17. Perdita Stevens, Rob Pooley. *Using UML: Software Engineering with Objects and Components*. Addison-Wesley, 2000.