On the Semantics of the Extend Relationship in Use Case Models: Open-Closed Principle or Clairvoyance?

Miguel A. Laguna, José M. Marqués, and Yania Crespo

Department of Computer Science, University of Valladolid, Valladolid
{mlaguna,jmmc,yania}@infor.uva.es

Abstract. A use case is a description of the interactions of a system with the actors that use it. The Achilles' heel of use cases is the unclear UML semantics, in particular the definition of the extend relationship. This article is an attempt to clarify the semantics of the extension mechanism. In particular, we advocate for the application of the open-closed principle, adding modification details in the extending use case, instead of in the base case. A revision of the UML standard would be impractical, but a disciplined reinterpretation of the *extend* and *extension point* concepts could represent a great improvement. Textual and graphical approaches (based in the UML Behavior meta-model) are considered. Using these recommendations, the base use cases can be independently described, while the extending use cases will be self-contained.

Keywords: use case, extend relationship, extension point, UML meta-model.

1 Introduction

Ivar Jacobson proposed use cases and incorporated them into his OOSE development method [13], being recognized as a useful technique to elicit and record user requirements. A use case describes the possible interactions that can occur between an actor and the future system. It also describes the responsibilities of the system under design, without getting into implementation techniques or system internal details.

The inclusion of use cases in the first versions of UML as standard modeling language [20] is probably the main reason of its widespread acceptation. Nowadays use cases are one of the preferred techniques for the representation of user requirements (and the external behavior of an information system considered as a black box). They are essential in the Unified Process, as this development method was evolved from the ideas of Jacobson [12]. One of the major controversies is the UML's explanations of *include* and *extend* relationships. In particular, *extend* concept remains vague, and apparently contradictory. Precise and unambiguous definitions are missing in the numerous UML documents. The original motivation of the *extend* relationship was the aspiration of never touching the requirements of a previous version of a system. But the presence of the *extension point* concept in UML violates the original purpose. In UML you must anticipate in the base use case those places at which extensions are permitted (i.e, the extension points). Therefore, UML's explanations for *extend* relationship are still subject to debate. Some conferences have been devoted to these and other conflicting aspects [7].

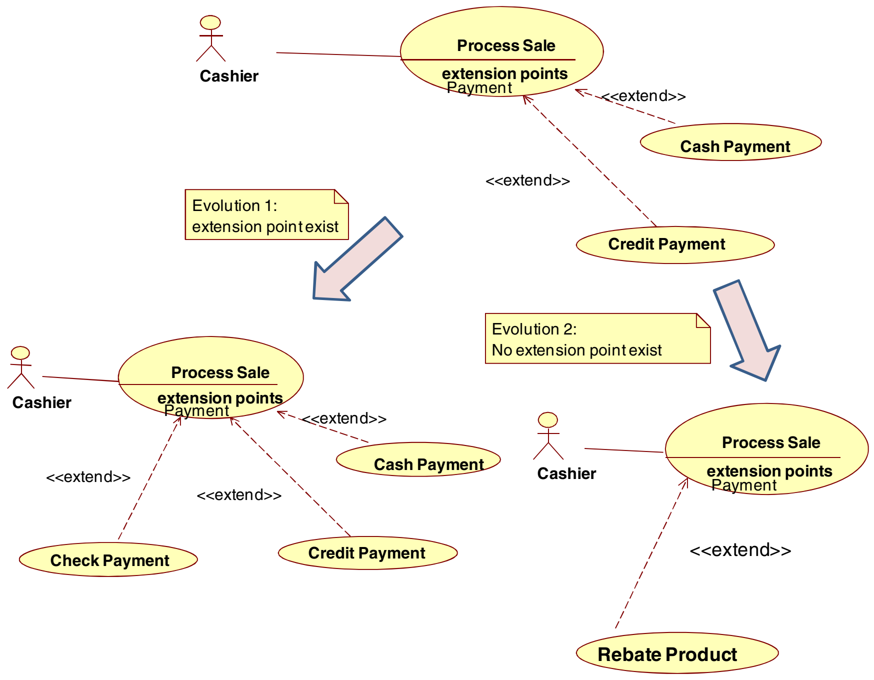

Fig. 1. UML extension (only possible if an extension point exists) and unanticipated extension (not possible in UML)

Figure 1, inspired in the example of *Point-Of-Sale* used by Larman [15], shows the differences between the original intention and the UML concept of *extension*. While the original extension could be considered as an enhancement of existing requirements (leaving untouched the original version), as depicted in evolution 2 of Figure 1, UML requires a previously defined extension point to incorporate a new extension. In the example of the base case of Figure 1, a new payment method ("*Check payment"*) is possible in UML but not the evolution 2 of Figure 1. In this situation, the system evolution requires a new use case that allows the application of discounts to certain products, as introduced by the cashier. If "*Process Sale*" does not have an adequate extension point, UML forbids the direct extension. The original intention is clearly more useful and we suggest recovering it in the UML context. As the alteration of UML meta-model is not an easy mission and we want to use conventional CASE tools, we propose to use some of the existing UML mechanisms to incorporate the original semantics of *extend*.

The rest of the paper is as follows: The next section briefly summarizes the UML vision of the *extend* relationships. Section 3 discusses the problems with the *extend* relationship, under the prism of the open-closed principle, and proposes a semantic reinterpretation of the *extension point* concept. Section 4 uses the UML Activity Package elements to visually represent an unanticipated extension. Section 5 presents related work and section 6 concludes the paper and proposes additional work.

2 The Extend Relationship in the UML Documentation

A use case describes an interaction between one or more actors and the system as a sequence of messages. Thus, a use case diagram has two types of nodes: actors and use cases, connected by association relationships. The original proposal of Jacobson also included two kinds of relationships between use cases: The *uses* and *extends* relationships, both indicated with generalization arrows. This syntax was initially preserved in primitive UML versions [20] but, beginning with the refined 1.3 version, a new set of relationships was proposed and this definition has essentially been kept, with minor changes, until the actual UML 2.1.2 version. From UML 1.3, relationships between use cases can be expressed in three different ways: generalization, *include*, and *extend* relationships. An *extend* relationship defines those instances of a use case that may be augmented with some additional behavior defined in an extending use case. While, the semantics of *include* relationship has always been reasonably clear, the *extend* relationship has generated a lot of controversy. Several modifications have been added to the different versions of UML. Attempts at removing these difficulties have been proposed in these documents. From here until the end of the article, we base the discussion on the official UML documentation, version 2.1.2 [19].

In the UML 2.1.2 meta-model, *Actor* and *UseCase* are both *BehavioredClassifier*, which itself is a descendent of *Classifier*. As UML documentation states, the *extend* relationship specifies how and when the behavior defined in the extending use case can be inserted into the behavior defined in the extended use case (at one *extension point*). Two important aspects are: a) this relationship is intended to be used when some additional behavior can be added to the behavior defined in another use case; b) the extended use case *must be independent* of the extending use case.

Analyzing the meta-model (see the white meta-classes of Figure 3), the *extensionLocation* association end references the *extension points* of the extended use case where the fragments of the extending use case are to be inserted. An *extensionPoint* is an owned feature of a use case that identifies a point in the behavior of a use case where it can be extended by another use case. The *extend condition* is an optional *Constraint* that references the condition that must hold for the extension to take place. The notation for conditions has been changed in UML 2: the condition and the referenced *extension points* are included in a *Note* attached to the *extend* relationship.

Semantically, the concept of an "*extension location*" is left underspecified in UML because use cases "are specified in various idiosyncratic formats". UML documentation refers to the typical textual use case description to explain the concept: "The use case text allows the original behavioral description to be extended by merging in supplementary behavioral fragment descriptions at the appropriate insertion points". Thus, an extending use case consists of behavior fragments that are to be inserted into the appropriate spots of the extended use case. An extension location, therefore, is a specification of all the various (extension) points in a use case where supplementary behavioral increments can be merged.

The next sections are devoted to analyzing this relationship and the connected *extension point* concept, including its necessity. Then, in section 4, we make use of the UML *Behavior* meta-model package to visualize the notion of use case extension.

3 The Extend Relationship and the Open-Closed Principle

In this section, we try to answer a preliminary question: Is the presence of the *extension point* concept in the use case models really indispensable? From the point of view of the semantics of the dependence relationship, the mere presence of an *extension point* in the base use case is confusing. Removing (or perhaps reinterpreting) the *extension point* concept could be a way of avoiding many problems.

The first intention of a dependence relationship is to establish a directed relationship between an independent element (the base or extended use case) and a dependent element (the extending use case). Therefore, if the base use case must have no information a priori about its possible modification, the obligation of predetermining an *extension point* is contradictory.

Meyer, as long ago as 1988 coined the open-closed principle [16], that states:

- *"software entities (classes, modules, functions, etc.) should be open for extension, but closed for modification"*

This definition applied originally to the object-oriented programming languages. In this context, an entity can allow its behavior to be extended without altering its source code. *Open for extension* means that the behavior of the entity can be extended (we can make the module behave in different ways as the requirements change, or to meet the needs of new requirements). *Closed for modification* means that the source code of such an entity is not modifiable. The well known solution to this dilemma is the use of inheritance in object-oriented languages. This idea applies directly to the specialization relationship among classes in object-oriented designs and can also be adopted in requirements artifacts. In a wider vision, a broader definition can be stated. If we substitute the concept of entity by generic software artifacts, we can affirm:

- *"software artifacts (requirements, designs, implementations, test cases, etc.) should be open for extension, but closed for modification"*

In the use case context, *open for extension* means that the behavior of the use case can be extended (we can make a use case behave in different ways as the requirements change, or to meet the needs of new requirements). *Closed for modification* means that the details of this use case (the textual or graphical description of the behavior) are not modifiable. We are ready to map the open-closed principle from object-oriented programming languages to use cases. We will try to apply the same approach in two parallel situations:

- A base class is extended by redefinition of an operation in a specialized class (the simple addition of a new operation is a less problematic situation)
- A base use case is extended by an extending use case

Figure 2 presents two situations that share the idea of extension: In a typical Sales application, a previously existent (cash) payment is extended by a new type of payment. In structural models we will need probably one or several new classes. We will need also to redefine the **`pay()`** operation, adding details of the credit card, authorization codes, etc. In the use case diagrams we solve the situation with a *extend* relationship and a new use case that incorporates the new steps (including the introduction of the credit card number, the connection with and external authorization service, etc.).

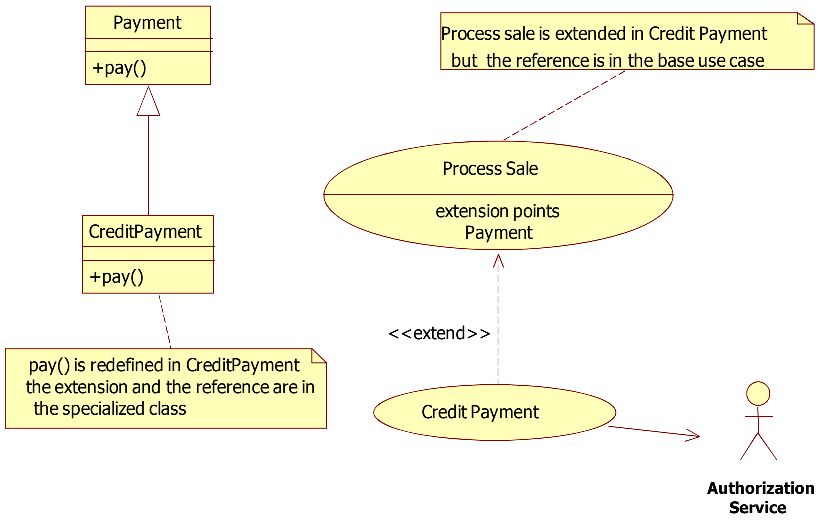
Payment
+pay()
CreditPayment
+pay()
pay() is redefined in CreditPayment the extension and the reference are in the specialized class
Process sale is extended in Credit Payment but the reference is in the base use case
Process Sale
extension points
Payment
<<extend>>
Credit Payment
Authorization Service

Fig. 2. The *extend* and *specialization* mechanisms in UML 2.1.2 [19]

If we consider Figure 2, we can appreciate that all the extension details of a class in a structural design diagram are defined in the extending class: The original Payment class in unaware of the redefinition of the pay method in the new class. The Eiffel version is absolutely clear (Java or C# versions are similar but not so explicit):

```
Class Payment
    feature
      pay(…) is do…end
    end
```

```
Class CreditPayment
        inherit Payment
           redefine pay end
        feature
          pay(…) is do…end
        end
```

Nothing in the original **Payment** class make reference to a hypothetical future modification and the class is closed. The two basic elements of the extension are the specialization relationship itself (and this is associated with the source element, i.e. the new class *inherit* the old) and the reference to the operation that need modification (and this is indicated by the *redefine* clause in Eiffel, or simply by duplicating the original name of the operation in UML diagrams).

The parallel version in the use case diagram is different: the details of the extension are in the new use case but, and here is the difference, an extension point must be previously defined in the original use case (in disagreement with the enunciated open-closed principle).

The open closed principle is feasible in object-oriented programming languages because they provide the adequate mechanism: the extension point (really a reference to the redefined method) and the extension itself (the new version) are both in the specialized class. The base class is unaware of the extension. Nevertheless, in use cases the extension is described in the new use case but the point of insertion is defined in the base class. The situation is loosely similar to the old C++ implementation of inheritance using *virtual functions* (the *"clairvoyance principle"* as opposite to the open-closed principle).

In the context of use cases, the situations we want to solve are, for example: a use case can evolve during the development of several versions of a software system; the requirements can change; new constraints or business rules can appear, etc. The essence of these situations is that the evolution usually occurs *"in an unexpected way"*. While the user requirements are being elicited, we have a possible solution with plain use cases: add an alternative sequence of steps to the set of exceptions of the use case, referring to a step of the main scenario.

The generalization of the idea is exactly the extension concept, useful when a) the use case is already completely developed through a collaboration that involves analysis or design models, or b) the complexity of the steps that must be added recommends separating this piece of behavior in a new use case. In both cases, as in the plain solution, we must be able to indicate where the new sequence must be inserted (after the original step n) and where the original scenario must continue (at the original step m). This can be as complex as needed, as in the idea of *extension points* with several fragment insertions.

The concept of scenario, as sequence of steps, is not directly present in the UML meta-model Use Case Package, probably in order to allow different particular implementations of the *Behavior* meta-class (visual or textual, formal, structured or informal). However, independently of the concrete format, the concept of *sequence of steps* could be present in this Use Case Package.

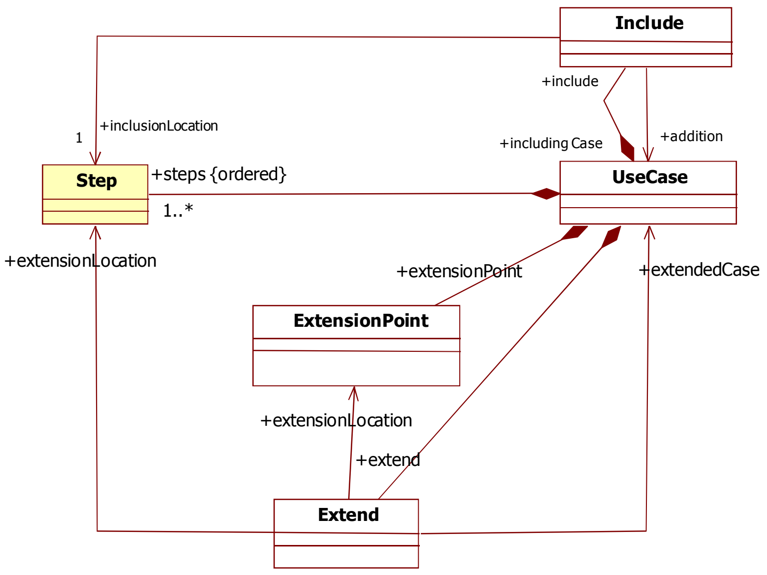

Fig. 3. The *Step* concept added to the UML Use Case Package meta-model

Consider provisionally the minimal variation of the meta-model shown in Figure 3: Only a new meta-class is added, representing the concept of *Step* and a *UseCase* owns an ordered set of steps. (A more complete version, considering all the possible alternative sequences, would include the intermediate *Scenario* concept as a set of ordered steps, owned by a *UseCase*.) This simple addition would allow defining the *extension* directly in function of the steps of the base use case, instead of a predefined extension point. As collateral effect (beneficial in this case), the exact position where the *Include* relationship is located can be made visible.

As we do not foresee immediate changes in the UML meta-model, we suggest an apparently incorrect solution to deal with this problem. Taking into account that the *Step* and *ExtensionPoint* definitions in Figure 3 are very similar from the viewpoint of the *Extend* meta-class, we consider that a use case has a set of steps (or sequences of inseparable steps) called *extension points*. If we think this way, all the steps of a use case are extensible. This interpretation implies that the use cases are completely open to future extensions (in the same way an unaffected class can be extended by a new one using inheritance in object oriented languages).

Really, this discussion about the meta-model is only conceptual. The details are in the textual step-based description of the use cases. In fact, this proposal assumes that the *extension point* concept is not used in the diagrams. In its place, we must indicate in the textual documentation of the extending use case:

a) The base use case that is extended.

b) The step where the extended use case is modified, using the same conventions of the alternative/exception fragments of the monolithic use cases; in other words, the precise step number must be referred.

c) The "rejoin point" in the extended use case in order to continue with the normal sequence of steps. The last step of the extension must indicate it.

The adoption of this approach means that all the possible situations must be documented in the textual information of the extending use case. The extended use case remains unchanged and unaware of the extensions. All the exceptions related with the new extension must be treated (and solved) in the new use case.

Summarizing the idea, in many cases (in particular in agile developments), it is preferable not to use *extension points* with the original UML semantics. Or, changing the point of view, all the steps of a use case can be considered as *extension points*. This version smoothes the learning curve of the technique by beginners (in fact we use this approach with our undergraduate students, avoiding many confusing discussions in the requirements gathering sessions).

In this Section we have dealt with the textual specification of the use case details. The next Section examines the alternative (and less informal) graphical representation of the extension notion using the behavioral concepts of the UML meta-model.

4 The Extend Relationship and the Use Case Behavior Specification

As visible in the meta-model of UML, a *BehavioredClassifier* (and hence a *UseCase*) has an associated *Behavior* that can have a set of *activities*, *actions*, *messages*, *states*, etc. The sequence of steps of the use case textual representation can be seen as a way of describing this behavior and many authors advocate for this perspective. This Section explores the possibilities that UML 2 have opened to specify the use case details in a graphical representation. The use of an *Interaction* as the representation of use case details is a familiar strategy. Larman [15] distinguishes the system sequence diagrams to specify the external behavior (requirements level) from the complete interaction that is the way the classes' new operations are discovered (the idea of collaboration as realization of the use cases).

The interaction diagrams have an evident problem: it is difficult to represent simultaneously all the alternative paths (in spite of the combined fragments possibilities). A better option is to use activities as the behavior specification of the use case. In the UML meta-model fragment of Figure 4 it can be realized that:

a) *Activity* is a subtype of *Behavior.*
b) *Activity* owns *ActivityNodes* (control or executable nodes)*, ActivityEdges,* and *ActivityGroups*.
c) *ActivityNode* or *Action* can be used as representation of a step of the use case scenarios. As UML states: "An action represents a single step within an activity, that is, one that is not further decomposed within the activity". Other specialized control nodes, such as *InitialNode* or *DecisionNode*, can be used to organize the alternative paths.
d) *ActivityPartition* (shown as a *swimlane* or a stereotype) can represent the actors that interact with the system (and the system itself). The *swimlane* graphical representation allows associating easily each *Action* with an actor or the system.

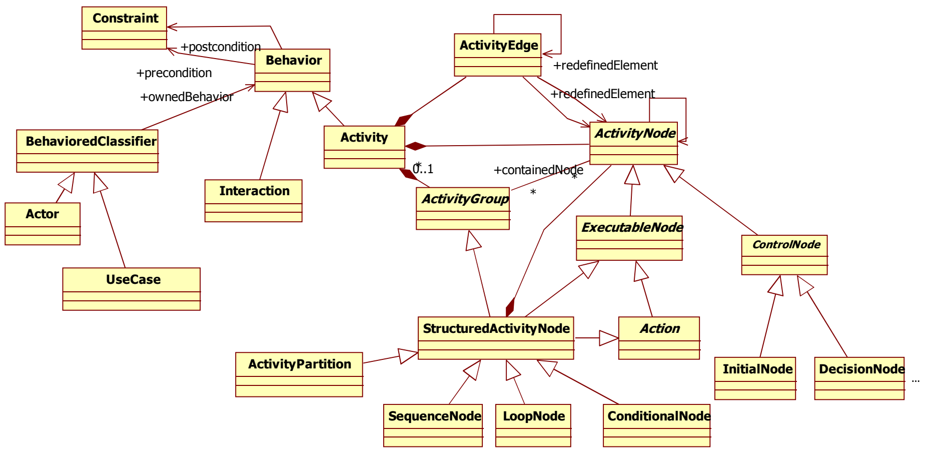

Fig. 4. Meta-model fragment of UML Activities and Use Case Packages

A fragment of the Process Sale use case (cash payment, following the example of Larman [15]) is shown in Figure 5. Only a successful scenario is represented and no extension points are provided.

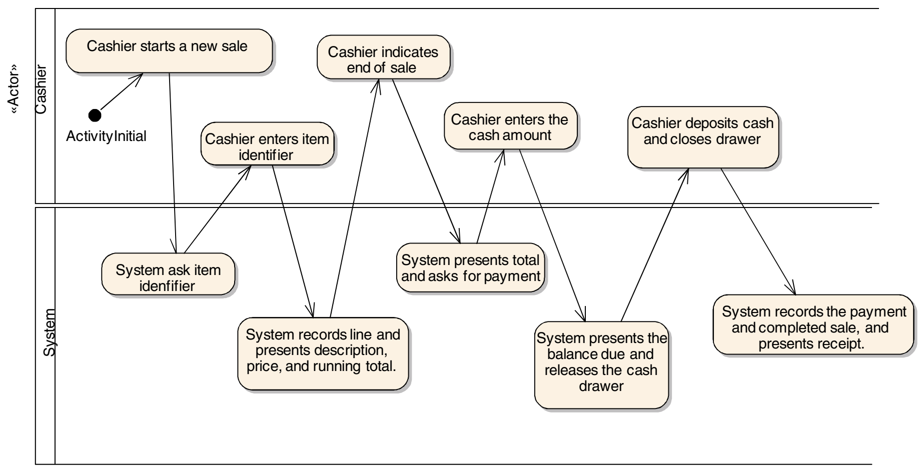
«Actor»
Cashier
System
ActivityInitial
Cashier starts a new sale
Cashier indicates end of sale
Cashier enters item identifier
Cashier enters the cash amount
Cashier deposits cash and closes drawer
System ask item idenfifier
System records line and presents description, price, and running total.
System presents total and asks for payment
System presents the balance due and releases the cash drawer
System records the payment and completed sale, and presents receipt.

Fig. 5. A simplified fragment (successful scenario) of the Process Sale use case of Figure 1

To handle the extension points, all the potential of the Activity Packages of Figure 4 must be used. Note that an *ActivityNode* can be redefined. This opens the possibility of replacing a node with a *StruturedActivityNode*, indicating that the original node is substituted by a group of nodes. The semantics of this element refers to a basic sequence of nodes (*SequenceNode* specialization) or a complex construction (*ConditionalNode* and *LoopNode*). The key is that *StructuredActivityNode* is simultaneously a specialization of *ActivityNode, ActivityGroup* (pag. 309 of [19]) and *Action* (pag. 310 of [19]). At the same time, *StructuredActivityNode* contains a set of *ActivityNodes* (pag. 309 of [19]), where each node can be itself a *StructuredActivityNode,* configuring a typical composite pattern.

The strategy for representing the extension mechanism can be seen in Figure 6: A stereotyped node represents the extension point in the base activity and each extending use case requires an activity that redefines the extension point node (<<extend>> stereotyped dependency). This interpretation is consequent with the UML semantics: a pre-defined "hook" allows inserting the new behavior at the extension point.

As in the previous section discussion, this is approach is only valid if we known *a priori* the details of the future extension. A set of alternatives are imaginable to conform to the open-closed principle. The original use case can be described with no mention to future extensions and the activity diagram will be similar to Figure 5. If a later evolution of the system requires an alternative payment form, two respectful variants of the previous solution are sketched in Figure 7. The first solution uses the generalization relationship between use cases, adding the extension point to the specialized use case. This variation does not change the original use case and divides the new information between the two new use cases: the intermediate *Advanced Process Sale* and the extending *Credit Payment.* However, the UML generalization semantics implies lost of control over the changes in the new version. The second variant uses the *package merge* mechanism of UML 2: The new version package contains a use case named as the original but with an added extension point. The package merge

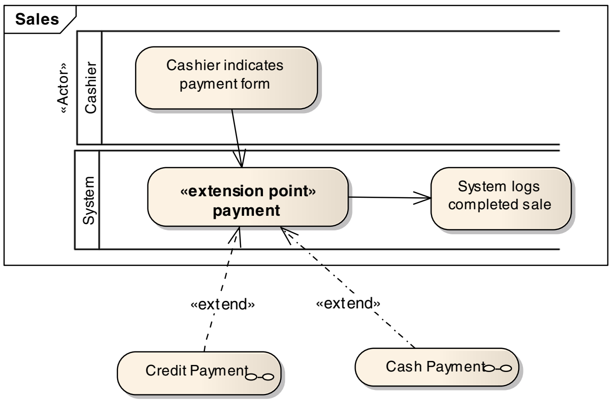
Sales
«Actor»
Cashier
Cashier indicates payment form
System
«extension point» payment
System logs completed sale
«extend»
«extend»
Credit Payment
Cash Payment

Fig. 6. A variant of the Process Sale use case with an extension point and two extensions

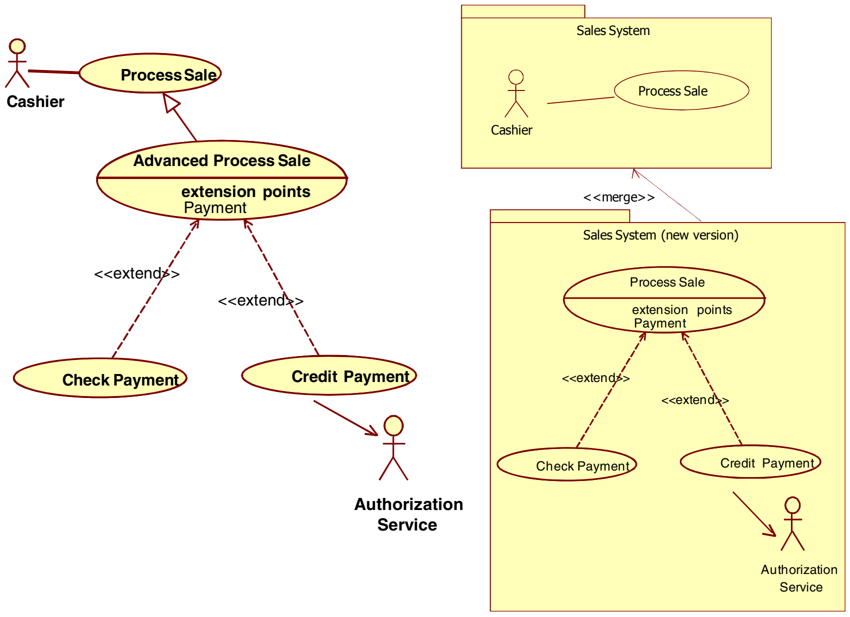

Fig. 7. The Use Case Process Sale and two reinterpretations of the Process Payment extension as an *a posteriori* addition

operation yields a full extensible use case. This is a valuable solution for the developing of product families, as new features can be incorporated gradually.

Although these solutions are semantically correct and complies with the open-closed principle, the overload is manifest. A slightly modification of the scheme of Figure 6 can exploit the node redefinition mechanism of the UML meta-model. Figure 8 shows how an activity node (not an extension point) can be replaced by a structured activity (<<redefine>> stereotyped dependency). The simplicity of the solution is appealing but it is only applicable to situations where all the sequence of elements can be neatly inserted in the position of the original node. Therefore, several problems remain unsolved: How to insert a new sequence when there is not a clear-cut node to be substituted? And, what if there are one or more "rejoin points" different of the extension point? Note that, if it is difficult to anticipate an extension point, it is yet more difficult to foresee the several possible "rejoin points" as the extension can include several success and fail scenarios

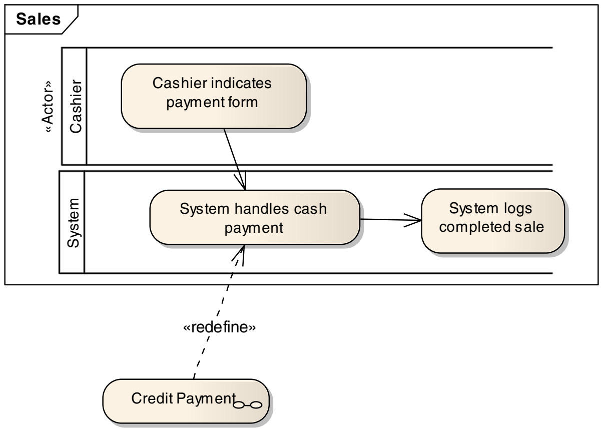
Sales
«Actor»
Cashier
Cashier indicates payment form
System
System handles cash payment
System logs completed sale
«redefine»
Credit Payment

Fig. 8. Activity node replaced by a structured activity

The solution emerges if we recognize that an extension is really an alternative path (group of elements or *ActivityGroup*) that complements another existing path (another group). Therefore, the extend relationship is better formulated between the original unchanged activity and the extension activity. Inside the new activity, the connecting points must be referenced. The concrete graphical representation is an open issue, and varies with the concrete facilities of CASE tools. Taking into account these considerations, the use cases behavior specification could be depicted as shown in Figure 9, where an activity ("Process Sale") is extended by another activity ("Rebate Product"), in parallel with the use case diagram. The inner region delimits the extension and the nodes outside this region are references to the original activity nodes that connect the old and new paths. The use of the associated Note indicates the extend condition (as in the standard) and the list of parameter values that serve to reference the connection points. This parameterization allows the reuse of the extending use cases with different base cases. The main advantage is the exclusion of the extension details from the original use case. In this sense, the final conclusion is that the UML meta-model provides the extensibility elements we need to apply the open-close principle in use case evolution.

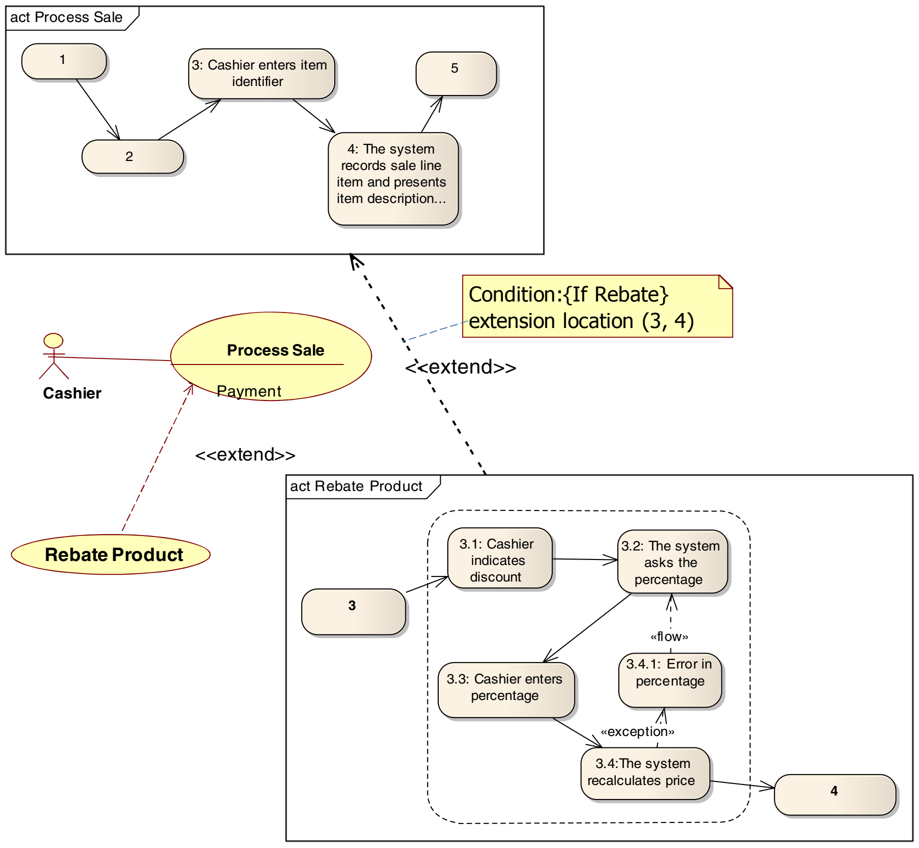
act Process Sale
1
2
3: Cashier enters item identifier
4: The system records sale line item and presents item description...
5
Condition:{If Rebate} extension location (3, 4)
<<extend>>
Cashier
Process Sale
Payment
<<extend>>
Rebate Product
act Rebate Product
3
3.1: Cashier indicates discount
3.2: The system asks the percentage
«flow»
3.3: Cashier enters percentage
3.4.1: Error in percentage
«exception»
3.4:The system recalculates price
4

Fig. 9. Not anticipated use case extension with inclusion of the connection points

5 Related Work

Many suggestions for modification of the UML meta-model have been proposed, including the use of ontologies instead [9]. Some authors, such as Berard [1] or Simons [24] have been detractors of use cases, thought Simons (in a work with van der Berg [27]) proposed control flow semantics for use case scenarios. Conversely, there are many works that try to improve or at least clarify them, such as the classical book of Cockburn [4] or the work of Williams *et al.* [29]. Metz et al. distinguish several textual variants for clarifying the semantics of extension points and rejoin points [17].

Many authors, such as Cockburn [4], have suggested that the most important characteristics of use cases are the textual details to be discussed with the end users while neglecting the visual representation and semantics proposed by UML. Others, such as Rumbaugh and Jacobson, insist on the graphics aspects [23].

Some additional relationships between use cases have been proposed. Rosenberg [21] uses precedes and invokes constructs to factor out common behavior. Conversely, other authors such as Larman [15] advocate not using the extend relationship or using only when it is undesirable to modify the base use case.

The *BehavioredClassifier* specialization of the use cases has been analyzed in [7]: The *Behavior* meta-class is a specification of how its context classifier (use case) changes over time and the *BehavioredClassifier* is a classifier that can have behavior specifications. In other words, a *BehavioredClassifier* is rather an ordinary classifier that can own behaviors [7]. The conclusion is that the formalization of use cases as classifiers in UML has obscure points: Two contradictory notions of use cases coexist in UML 2: "set of interactions" vs. "set of entities". The authors propose the meta-model should be changed to make *UseCase* a subtype of *Behavior*, not of *BehavioredClassifier*. Alternatively, they admit that the meta-model may be kept as it is, but it should be recognized that a use case is the specification of a role. Williams *et al.* also analyze the UML 2 meta-model and propose changing *UseCase* to a subclass of *Behavior* [29]. Isoda states that UML 2 has a correction about the relationship between use cases and actors, which effectively means that UML has finally abandoned the idea of "actors call operations of a use case", but the details of UML 2 in fact still retain those defects [10]. Jacobson believes that integrating use cases and aspect oriented programming (AOP) will improve the way software is developed. The idea is to slice the system and keep the use cases separate all the way down to the code. "In the long term we will get more of extension-based software-extensions from requirements all the way down to code and runtime; and extensions in all software layers, for example, application, middleware, *systemware*, and extensions across all these layers" [11]. Other approaches different from UML have considered the behavior extension. For example, the Taxis semantic model defines transactions as class hierarchies [18].

Sousa *et al.* adapt use-cases in order to explicitly provide the separation of crosscutting concerns in requirements artifacts. They propose *crosscutting* use-cases to separate crosscutting behavior (for example, security) from the main scenario [26]. In the same line, Somé and Anthonysamy [25] present an approach for modeling use cases with aspect-oriented techniques. Cross-cutting requirements are modeled in addition to functional requirements using a new relationship stereotyped as <<*aspect*>>. The composition is based in Petri nets properties. In both cases, new elements are proposed to complete the UML meta-model or to represent the details of use case composition.

Araujo *et al.* compose aspectual and non-aspectual scenarios instead use cases. Non-aspectual scenarios are modeled as UML sequence diagrams. Aspectual scenarios are modeled as Interaction or State Machine Pattern Specifications [6]. Then, aspectual and non-aspectual scenarios are composed at sequence diagram and state machine levels [1]. Whittle proposes use case charts with the aim of formally defining the semantics of scenarios [28]. These proposals are close to our vision of composing fragments of models but introduce additional complexity and separates from UML standard.

As a general observation about these advances, in spite of the utility of considering early aspects (for example to model repetitive requirements as security or audit steps), we think that extension problems should not be treated inevitably as aspects.

Braganza *et al.*, discuss the semantics of use case relationships and their formalization using activity diagrams in the context of variability specification. They propose an extension to the *extend* relationship that supports the adoption of UML 2 use case diagrams into model driven methods. The proposal results from the 4 Step Rule Set, a model driven method in which use cases are the central model for requirements specification and model transformation [3].

The common conclusion of most of the work done in use case semantics is that the question is not well solved in UML and a redefinition of the concepts is needed. We believe that our contribution can help in this redefinition.

6 Conclusions and Future Work

In this article, the problems of interpretation of the *extend* semantics in use case models are analyzed. An improvement of the *extension* notion is proposed, based in the open-closed principle, by adding the *Step* concept or alternatively using as substitute the *extension point* concept itself.

The second part of the article presents a possible graphical approach, using the Behavioral aspects of the UML meta-model. We have shown that UML provides the extensibility elements we need to apply the open-close principle in use case extension. We think that, without neglecting major future modifications in the UML meta-model, these proposals can help in the process of elicitation and specification of functional requirements, clarifying the intention of the final users.

In conclusion, we believe that the set of semantic reinterpretations proposed in this article can help to solve many of the practical extension problems the requirements engineers face in their daily work. The empirical validation is a work in progress to check the usefulness of the approach. We are defining a set of problems so that diverse groups (from undergraduate students to experts) can use these techniques to compare (via controlled experiments) the comprehensibility and feasibility of the diverse variants of the extension concept.

References

1. Araujo, J., Whittle, J., Kim, D.: Modeling and Composing Scenario-Based Requirements with Aspects. In: Proceedings of the 12th IEEE international Requirements Engineering Conference (2004)
2. Berard, E.V.: Be Careful with Use Cases. Technical report. The Object Agency, Inc. (1995)
3. Braganca, A., Machado, R.J.: Extending UML 2.0 Metamodel for Complementary Usages of the «*extend*» Relationship within Use Case Variability Specification. In: Proceedings of the 10th international on Software Product Line Conference, pp. 123–130. IEEE Computer Society, Washington (2006)
4. Cockburn, A.: Writing Effective Use Cases. Addison-Wesley Professional, Reading (2000)
5. Constantine, L., Lockwood, L.: Software for Use. Addison-Wesley, Reading (1999)
6. France, R., Kim, D., Ghosh, S., Song, E.: A UML Based Pattern Specification Technique. IEEE Transactions on Software Engineering 30(3), 193–206 (2004)
7. Génova, G., Llorens, J., Metz, P., Prieto-Díaz, R., Astudillo, H.: Open Issues in Industrial Use Case Modeling. In: Jardim Nunes, N., Selic, B., Rodrigues da Silva, A., Toval Alvarez, A. (eds.) UML Satellite Activities 2004. LNCS, vol. 3297, pp. 52–61. Springer, Heidelberg (2005)

8. Génova, G., Llorens, J.: The Emperor's New Use Case. Journal of Object Technology 4(6), 81–94 (2005); Special Issue: Use Case Modeling at UML 2004
9. Genilloud, G., William, F.: Use Case Concepts from an RM-ODP Perspective. Journal of Object Technology 4(6), 95–107 (2005); Special Issue: Use Case Modeling at UML 2004
10. Isoda, S.: A Critique of UML's Definition of the Use-Case Class. In: Stevens, P., Whittle, J., Booch, G. (eds.) UML 2003. LNCS, vol. 2863, pp. 280–294. Springer, Heidelberg (2003)
11. Jacobson, I.: Use Cases and Aspects—Working Seamlessly Together. Journal of Object Technology (2003), `http://www.jot.fm`
12. Jacobson, I., Booch, G., Rumbaugh, J.: The Unified Software Development Process. Addison-Wesley, Reading (1999)
13. Jacobson, I., Christerson, M., Jonsson, P., Overgaard, G.: Object-Oriented Software Engineering, A Use Case Driven Approach. Addison-Wesley, Reading (1994)
14. Jacobson, I., Griss, M., Jonsson, P.: Software Reuse. In: Architecture, Process and Organization for Business Success, ACM Press/Addison Wesley Longman (1997)
15. Larman, C.: Applying UML and Patterns: An Introduction to Object-Oriented Analysis and Design and the Unified Process, 3rd edn. Addison Wesley, Reading (2004)
16. Meyer, B.: Object Oriented Software Construction. Prentice-Hall, Englewood Cliffs (1988)
17. Metz, P., O'Brien, J., Weber, W.: Specifying Use Case Interaction: Clarifying Extension Points and Rejoin Points. Journal of Object Technology 3, 87–102 (2004)
18. Mylopoulos, J., Bernstein, P.A., Wong, H.K.: A language facility for designing database-intensive applications. ACM Trans. Database Syst. 5(2), 185–207 (1980)
19. OMG, Unified Modeling Language: Superstructure, version 2.1.2. formal doc. 2007-11-01. 2007 (2007)
20. Rational Software Corporation Unified Modelling Language Version 1.1 (1997)
21. Rosenberg, D., Scott, K.: Applying Use Case Driven Object Modeling with UML: A Practical Approach. Addison Wesley, Reading (1999)
22. Rumbaugh, J., Blaha, M., Premerlani, W., Eddy, F., Lorensen, W.: Object-Oriented Modeling and Design. Prentice Hall, Englewood Cliffs (1991)
23. Rumbaugh, J., Jacobson, I., Booch, G.: The Unified Modeling Language Reference Manual, 2nd edn. Addison-Wesley Professional, Reading (2004)
24. Simons, A.J.H.: Use Cases Considered Harmful. In: 29th Conf. Tech. Obj-Oriented Prog. Lang. and Sys. (TOOLS-29 Europe), pp. 194–203. IEEE Computer Society, Los Alamitos (1999)
25. Somé, S.S., Anthonysamy, P.: An approach for aspect-oriented use case modeling. In: Proceedings of the 13th international Workshop on Software Architectures and Mobility, pp. 27–34 (2008)
26. Sousa, G., Soares, S., Borba, P., Castro, J.: Separation of crosscutting concerns from requirements to design: Adapting the use case driven approach. In: Proceedings of the Early Aspect Workshop at AOSD, pp. 93–102 (2004)
27. van den Berg, K.G., Simons, A.J.H.: Control flow semantics of use cases in UML. Information and Software Technology 41(10), 651–659 (1999)
28. Whittle, J.: Precise specification of use case scenarios. In: Dwyer, M.B., Lopes, A. (eds.) FASE 2007. LNCS, vol. 4422, pp. 170–184. Springer, Heidelberg (2007)
29. Williams, C., Kaplan, M., Klinger, T., Paradkar, A.: Toward Engineered, Useful Use Cases. Journal of Object Technology 4(6), 45–57 (2005); Special Issue: Use Case Modeling at UML 2004