What Is This Thing Called Use Case Inheritance?

Pierre Metz[1]

[1] Brose Fahrzeugteile GmbH & Co. KG, Germany
(formerly Cork Institute of Technology, Computing Dept., Cork, Ireland and Darmstadt University of Applied Sciences, German)
pierre.metz@brose.com, pierre.metz@intacs.info

Abstract.
In more than two 2 decades of use case modelling there has been a imprecisely defined notion in UML since v1.1 that has never been fully understood, namely use case inheritance (UCI). Therefore, this paper suggests a necessary reconciliation to achieve a broader acceptance and attractiveness in practice while reducing confusion, with a clear demarcation from the Include/Extend relationships.
This is done based on implications from the author's completed PhD research, UCI suggestions found in research contributions, technical text books as well as literature about OO inheritance semantics, and the author's personal professional industry experience. Rather than being a typical formal research paper, the drivers of the presented solution proposal are to offer pragmatic and practical UCI application rules for the industry. This should offer a basis for further qualitative validation by requirements engineers in practice, and, also for future conceptual research.

Keywords: Use Cases, Use Case Relationships, Requirements Engineering, Inheritance, Specialization, Generalization, Subtyping, Polymorphism, UML.

1 Recollection of Use Case Basics

1.1 Actors, Goals, and Use Cases

An actor specifies a role that can be taken by a person, a piece of hardware, a component, or a software application [18],[17],[20]. Each actor has certain operational responsibilities imposed by the surrounding business processes and rules. In order to fulfil its responsibilities, the actor has to perform a number of operations. It wants some subset of these operations to be facilitated by a software application or hardware

apparatus. Thus, it sets corresponding goals for the system to deliver. These goals lead to desired functional system requirements specifications, i.e. not internal design solutions, expressed by use cases [1],[10]. Each use case delivers a single go al (see Example 1). An
actor instance processes only those use cases that the actor is connected to.

Example 1: Use Case Goal
"Register New Customer Order"

Basic Course:

```
1. Sales clerk enters customer
ID.
2. System displays customer pro-
file.
3. Sales clerk confirms that the
customer's credit rating is suf-
ficient.
4. System assigns an order ID.
5. Sales clerk registers the
desired trade items and payment
information.
```

Use Case Postconditions:

```
System has initiated an order,
has documented payment infor-
mation, and has registered the
order with the customer.
```

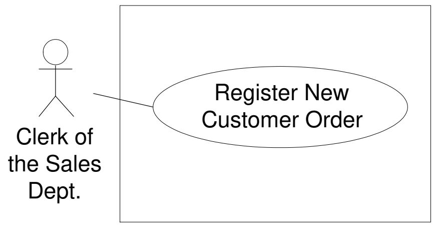

Fig. 1: Use case diagram for Example 1

Alternative Courses:

```
1a. Sales clerk wants to look up
    the customer:
     .1 System shows all custom-
        ers.
     .2 Sales clerk browses the
        list and selects one.
  Rejoin at 2.

3a. Customer's outstanding debts
     are above the threshold:
     .1 System notifies the key
       account manager for media-
       tion purposes.
  Use case fails.
```

1.2 Use Case Pre- and Postconditions

From the goal of each use case a set of corresponding outcomes is derived to be established upon successful goal delivery [1],[10], i.e. successful use case completion. Each of these results is required by the business processes associated with the discussed use case and, therefore, is delivered to at least one primary actor or stakeholder [1],[4],[5],[10],[19],[20],[25]. The set of use case business results are also called *use case postconditions* [4],[5],[20],[30], In many cases, a use case requires some condition to hold *before* it can be triggered and executed by an actor instance. These are called *use case preconditions*. The view of use case goals, use case pre- and postconditions were considered a "contractual" specification [10],[20],[31] thereby seemingly resembling the Design-by-Contract principle in [25]. However, there is a fundamental difference: Design-by-Contract demands the caller of a service to guarantee the preconditions in order for the callee to deliver this service, the result of which is specified by postconditions. Clearly, for use cases an actor instance is the "caller". However, use case precondition checking is always a system responsibility, i.e. done by the callee but never by the caller [1],[4],[5],[10],[20],[30]. For example, the preconditions of an ATM's "Withdraw Cash" use case would include "Cash reservoir not empty". Therefore, use case pre- and postconditions do not correspond to Design-by-Contract.

1.3 Use Case Interaction

Accomplishing the goal-driven postconditions by the system might require active interaction with the actor instance at the *system boundary* [1],[4],[10],[17],[18],[19],[20]

1.4 The Extend- and Include-Relationships

The Include and Extend relationships have been defined and explained as *static* relationships for use case *restructuring* or *refactoring* of *specifications*, e.g. for removing redundancy or use case decomposition [1],[4],[5],[15],[14],[18],[19],[22],[23],[24],[32]. In this respect, Extend is explained as attaching to a base use case a description of a set of interaction steps that can be subject to a condition. On the contrary, Include is understood as attaching a description of a mandatory set of interaction steps. Therefore, an inclusion/extension use case always remains a *part* of a *static* system functionality *description*. In fact, in [24] it has been shown that Include and Extend follow whole-part (aggregation) relationship semantics.

If factored out by Include/Extend these interaction parts also form a use case, i.e. they receive a goal and post-conditions. It follows that this goal is a sub-goal of the base use case, i.e. the summarising goal for the factored-out interaction which is a *subset* of the base use case interaction. This is obvious for Include-attached interaction, and it is also true extension (and therefore conditional) use cases. Arguing that an extension use case may or may not be executed depending on condition evaluation is a *runtime* or at least a *scenario* perspective; in contrast, as shown above use cases are *static requirements descriptions* (see Section 1.1) prior to design, so talking about goals is a *conceptual* and *business semantics* perspective.

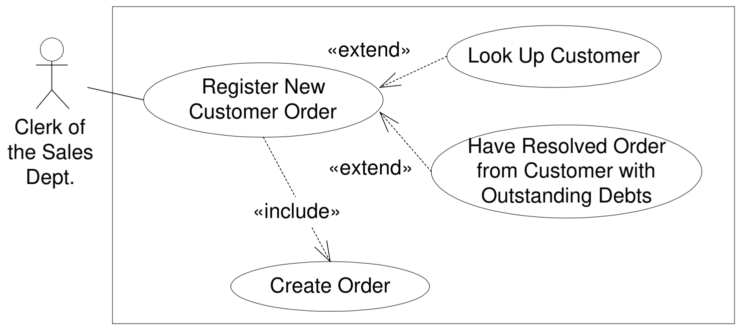
Clerk of the Sales Dept.
Register New Customer Order
«extend»
Look Up Customer
«extend»
Have Resolved Order from Customer with Outstanding Debts
«include»
Create Order

Fig. 2: Use case diagram of Example 1 showing the alternative courses, and interaction steps 4 and 5 factored out

Hence, exclusion use cases hold sub-goals, i.e. either something additional, or a different approach to fulfilling a base use case interaction step [23]. See Fig. 2 as an example: the goal “Create order” would be a summarising goal of the interaction steps 4 and 5 in Example 1; the goals for alternative course 1a in Example 1 would be “Look Up Customer”, and its postcondition could be “Customer has been marked”. We see that goal-subgoal semantics are independent of model restructuring/refactoring through Include/Extend [1],[10].

2 Use Case Inheritance – Literature Review and Related Work

In practice, and in some literature e.g. [20], it is sometimes believed that Extend can be seen as a generalisation relationship. However, Section 1.4 and, also, references [22],[24],[32],[36] show that this view cannot be uphold. Furthermore, the fact that UML [26] keeps Extend explicitly separate from UCI implies that Extend is not meant to have the same semantics since otherwise this distinction would be meaningless.

It can often be observed that it is silently assumed, or even explicitly claimed, that UCI works the "same way" as with classes in the OO domain [17],[18],[19],[26],[27]. This view is probably fostered by UML's persisting foundation of use cases as Classifiers since v1.1, which yields object semantics [26], and also by former UML v1.4's statement:

> *"A generalization relationship between use cases implies that the child use case contains all the attributes, sequences of behavior, and extension points defined in the parent use case, and participates in all relationships of the parent use case. The child use case may also … add additional behavior into and specialize … behavior of the inherited ones."* [26].

Most authors avoid making any commitment and prefer to provide explanations like UCI is about "variations", "indicating commonalities", the child use case "doing a bit more" than the parent use case, or "adding to and redefining/overriding" parent interaction and properties [1],[3],[4],[5],[6],[19],[20],[28],[29].

In [18] Jacobson introduces the idea of "abstract use cases", i.e. a use case which contains generic interaction step placeholder to be expanded on in a child use case, a concept which is also supported by UML [26].

In [4],[17],[18],[27] it is claimed that a child use case must preserve the parent use case order of interaction steps. Apparently, this is driven by the assumption that use case inheritance should ensure OO behavioural conformity when substituting child entities for parent entities (see Section 3 for more explanation).

In [2] it is suggested that

> *"inheritance between use cases should be applied whenever a single condition … would result in the definition of several alternative courses."*

thereby making fuzzy the concepts of locally specified alternative interaction courses (*ScenarioPlusFragments* use case pattern in [1],[10] and the extracting of such alternative courses via Extend (*PromotedAlternative* pattern in [1]). Why creating a new use case instead of simply adding the new behaviour to the given use case?

To [1] Rawsthorne contributed the *CapturedAbstraction* use case pattern, which develops further Cockburn's idea of documenting Technology Variations [10]. This pattern suggests placing pure technology variations of given parent use case (e.g.

interactions for blind people, voice control, loading of a keycard instead of money dispensing for a Withdraw Cash use case) into new child use cases instead of cluttering the parent use case up with a large number of alternative courses.

Cockburn [10], looking at use case inheritance from a Generalization perspective, disregards the concept in practice. The UML [26] use case inheritance semantics of allowing a child use case to be substituted wherever a parent use case is mentioned he explains as being in conflict with business process logic.

Without any connection to use case inheritance, he suggests removing redundancy through parameterisation: consider the use case "Find Customer Order" in the sales domain. Further consider the use case "Find Insurance Policy" in the insurance domain. Cockburn states that, though there are differences due to the different business domains, these use cases will contain identical interaction logic with respect to "finding something". He suggests placing all identical interaction steps into the parent use case and adding placeholders at those places at which each child use case sets concrete concern-specific interaction (see Example 3, below). A notation for this is not proposed though. The same suggestion is made in [5] by Bittner/Spence and [16] by Hitz/Kappel.

In [30] it is claimed, without further explanation, that there is no UCI at all.

3 Consulting Object-Oriented (OO) Inheritance Semantics

Basically, in the OO domain inheritance is not only considered an implementation tool but also a general modelling and "thinking" concept. It addresses the creating of abstractions based on existing abstractions without modifying the latter (*Open-Closed principle*). Ideally, inheritance ought to represent *subtyping* which is also referred to as *conceptual specialisation*, or *strict inheritance* [35]. Subtyping demands true semantic correspondence of the child to its parents. A further goal of subtyping is the enabling of the processing of an object of a subtype on behalf of an object of a supertype (*substitutability*), thereby demanding *behavioural compatibility*, a term which is also referred to as *semantic conformance* or *behavioural subtyping*. This introduces *dynamic polymporphism*, as also called *subtyping polymorphism* or *dynamic typing,* that is realised by *late-binding*. Such behavioural compatibility has been addressed by e.g. Meyer's Design-by-Contract [25], the Liskov Substitution Principle (LSP) [21], or by Cook/Daniels [11], all of which guarantee the Open-Closed principle.

Inheritance further introduces the possibility of defining semantic abstractions with the decision about their properties' data types being deferred to instantiation time. This is called *parametric polymorphism* which does not require runtime concepts like late-binding [8],[34]. Today, in the programming domain this concept is often compared to *template classes* or *generic programming*.

In spite of the fact that subtyping was often regarded as the only legitimate reason for applying inheritance [35], the evolution of OO approaches, systems, languages, and concepts has shown that inheritance does not necessarily ensure subtyping. Rather, inheritance allows the modification of child properties in various manners by adding, redefining/overriding, and even removing properties, and by changing visibility of properties [35]. In the implementation domain, this is particularly exploited for pragmatic reasons such as saving coding effort (reuse), reducing memory needs, or introducing more efficient algorithms, all of which is not based on conceptual reasons. Furthermore, inheritance can be used as a pure *hierarchical structuring* tool by introducing *abstract classes* revealing *abstract operations*, i.e. having no methods; in contrast, *concrete* sub-classes provide such methods, thereby further supporting polymorphism. Pure abstract classes, i.e. having abstract operations only, equal to the concept of *interfaces*. This, in turn, has led to the concept of *interface inheritance* as opposed to *property inheritance* [35], i.e. inheriting operations vs. inheriting methods, attributes, constraints and associations. Due to the possibility of object concatenation (*delegation*) [33],[35] the concept of *static* vs. *dynamic inheritance* was discussed, the latter of which provides the ability for an object to change its parents at runtime [8],[33],[35].

All this makes it impossible to make the objective statement that the "very essence" of all types and variations of inheritance apparently is allowing *incremental modification* while following the *Open-Closed principle* [35].

4 My Proposal – Discouraged Use Case Inheritance Semantics

4.1 Use Cases Cannot be Treated Polymorphically "At Runtime"

In agreement with Cockburn [10] I suggest that substitutability (see Section 3) should not valid for use cases. Why? Use cases are not programs but static requirements specifications (see Section 1.1). Hence, why would an actor instance need to process a more special or more general use case on behalf of the one that was specified for it based on its individual operational responsibilities, role definition, job description? Why would an actor instance perform a use case the postconditions of which would deliver more or even less than needed by the surrounding business process needs and business rules? Therefore, use case performances are *non-substitutable*, i.e. for use cases there neither is *runtime polymorphism* and *late binding* nor *dynamic inheritance*.

4.2 Multiple Use Case Inheritance Disregarded

UML allows multiple inheritance of use cases [26]. However, my opinion is that the idea of multiple UCI is not applicable because it violates the Separation of Concerns principle [12],[22]: as we know from Section 1.1 a use case is a goal-driven requirements specification of an *individual*, *independent*, and *behaviourally coherent* system functionality representing a system-supported part of *the pre-existing business processes*. In this respect, a use

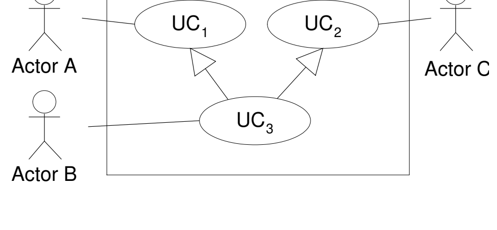

Fig. 3: Multiple use case inheritance

case is, in fact, a single business concern [22]. Consequently, the collapsing of two distinct business concerns, i.e. use cases, into one is in conflict with this principle (even if UC$_3$ in Fig.3 was adding further interactions). Would stakeholders and end users actually demand new a system functionality that consists of an assembly of two distinct already existing ones?

In fact, the existence of a use case child with multiple parents rather indicates the general confusing of inheritance and the employing of whole/part-like use case relationships, i.e. Include or Extend (see Section 1.4). A similar confusion has already been reported in the OO programming domain [34] where often *mixin classes* [7] are created where delegation, i.e. aggregating classes by associations, or single inheritance, respectively, would have been the appropriate tool [34].

5 My Proposal – Encouraged Use Case Semantics

5.1 Parametrization of Identifiers Within Interaction Steps ("Parametric Polymorphism")

I combine Cockburn's [10] notion of parameterized use cases, and the ideas of Bittner/Spence [5] and Hitz/Kappel [16], to suggest the idea of *parametric polymorphism* for use cases as follows:

Rule 1. Parameterizing Identifiers
Use case interaction steps contain identifiers (i.e. generic data placeholders) instead of concrete values, e.g. "customer name" instead of "John Doe". Now, for maximum reuse purposes and reducing redundancy, even such identifiers may be left unspecified (parameterization) in the parent, and in the child use cases only the expanding concrete identifier names have to be listed. The rest of the parent use case is valid also in the child use cases. E.g. a child use case identifier "customer name" could be abstracted to "search criterion" in the parent use case.

Example 2:

The following automotive domain example exploits the concept for propagating light signals to a hitch controller on a car driver's actions such as using the indicators or braking. In the parent use case generic identifiers are given in italics:

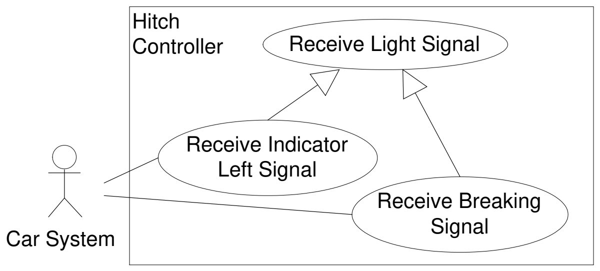

Fig. 4: Identifier parameterization

Goal "Receive Light Signal"

Trigger: `Edge change for <signal> detected`

Basic course:
```
1. Hitch controller either activates <actuators> upon rising edge
   or deactivates <actuators> upon trailing edge.
```

Postconditions:
```
 Electric contacts of <actuators> activated when corresponding
car system <signal> activated.
```

Goal "Receive Indicator Left Signal"

```
signal:     left indicator
actuators:  IndicatorRearLeft
```

Goal "Receive Braking Signal"

```
signal:      braking
actuators:   BrakingRearRight,
             BrakingRearLeft
```

5.2 Parametrization of Entire Interaction Steps ("Hierarchy Abstractions")

I adopt Jacobson et al.'s original idea of "abstract use cases" [18] (see also [27],[28]) Rawthornes *CapturedAbstraction* use case pattern [1], and Cockburn's *Technology Variations* [10] by integrating in terms of the following rule:

Rule 2. Interaction step set placeholders
Even an entire set of interaction steps can be replaced by a generic placeholder, i.e. not providing concrete behaviour ("abstractness", "virtual" interaction step). This indicates that a child use case will later expand on this placeholder by providing concrete interaction steps.

Example 3:

Fig. 6 shows a simplified example from a real world requirements specification for 4th generation (GSM-based, i.e. no hardwired online connections) EFTPOS terminals (electronic funds transfer at point of sale).

Goal "Make EFTPOS"

Basic course:
```
  1. Inserter inserts
     card and amount.
  2. System validates
     card information
     remotely.
  3. <interaction
     placeholder>
  4. System ejects
card.
```

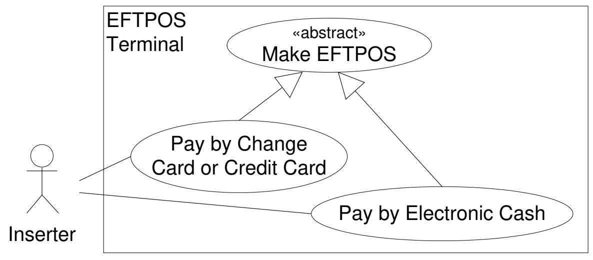

Fig.5: Use case inheritance for technology variations

Postconditions: `Card taken by inserter. Debit registered.`

Goal "Pay by Change Card or Credit Card"

Basic course:
```
At <interaction placeholder>
 1. System asks for electronic
    signature.
 2. Inserter signs with e-pen.
 3. System debits card holder's
    credit card account.
```

Goal "Pay by Electronic Cash"

Basic course:
```
At <interaction placeholder>
 1. Inserter enters PIN.
 2. System validates the PIN.
 3. System debits card
    holder's bank account.
```

Postconditions:

`Credit Card or Change Card taken by inserter. Debit registered with credit card account.`

Postconditions:

`Debit card taken by inserter. Debit registered with card holder's bank account.`

5.3 Specialization of Use Cases ("Property Inheritance", "Incremental Modification")

It follows from the general scientific principle of "Ockham's Razor" that it is is of little practical and scientific use if a new concept would only be an alternative to what can already be done with given modelling elements. Therefore, UCI semantics should be differentiable from Include and Extend in particular. Consequently, I discourage the suggestions in [2] (see Section 2). Further, I propose the following new detailed rules:

Rule.3 Strengthening Use Case Goals and Postconditions
This means that if Include and Extend carry *sub*-goals only then, in the spirit of the OO Design-by-Contract principle [25] and the LSP [21], UCI should be able to *either* maintain *or* strengthen the base use case goal; consequently, as postconditions must always support the goal, they need *either* to be held *or* strengthened, too.
Example 4 below shows a strengthening scenario, while in Example 3, above, the different wording of the goal and postconditions is only because of the different interaction step placeholder; from the business domain viewpoint they are actually equivalent.

Rule.4 Modifying Interaction Steps [1],[5],[19],[20],[28],[29]
In Section 2 we have seen that some authors claim that a child use case must not redefine the parent use case's order of interaction steps; apparently, this is to suggest that UCI shall ensure OO behavioural conformity when substituting child instances for parent instances. However, Section 4.1 explains why OO behavioural conformity should not be required for use cases. Further, from Section 1.3 we understand that interaction steps "connect" the use case goal with the postconditions. It thus appears that what solely governs the design of interaction steps are business rules (business domain semantics) and functional system requirements. For these reasons I suggest allowing a child use case to reorder and modify inherited parent use case interaction steps, and to add new ones. Correspondingly, guards of alternatives course may be adapted, and their branching points relocated, by the child use case. Further, inherited inclusions or extensions might need to be dissolved because former redundancy might vanish, or new inclusions or extensions ones introduced because of new redundancies. In a use case diagram, the dropping of a base use case's Include/Extend is represented simply by graphically not repeating them for the child use case. However, there are two constraints: any such modification must ensure consistency with the underlying business domain rules, and, also, must never cause the parent use case postconditions *and* the parent use case goal be *weakened* (i.e. Rule 3 shall apply). Let us look at Example 4 demonstrating the application of these rules.

Example 4: Applying the above rules 3 and 4 to Example 1

- *Rule 3:* the goal "Register New Customer Order with Specials" is a stronger goal for "Register New Customer Order" (details given in Example 1) because the former adds price reduction for VIP customers. This is reflected correspondingly by the stronger child use case postconditions, i.e. the child establishes everything the parent does plus the recording of negotiated price reduction;

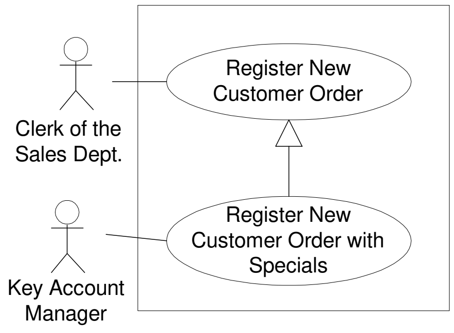
Register New Customer Order
Clerk of the Sales Dept.
Register New Customer Order with Specials
Key Account Manager

Fig. 6: Use case reuse

- *Rule.4:* correspondingly, the child use case goal reveals the new interaction step at label 6. It is also decided that the child use case waives the inherited step 3 (indicated by strikethroughs) as a VIP customer shall be attended to irrespective of their obligations. Correspondingly, the inherited alternative course 3a is no longer applicable either.

Goal "Register New Customer Order with Specials"

Basic Course:

```
1. Key account manager enters customer ID.
2. System displays the customer profile.
```
~~3. Sales clerk confirms that the customer's credit rating is sufficient.~~
```
4. System assigns order ID.
5. Key account manager registers the desired trade items
6. Key account manager grants price reduction.
```

Alternative Courses:

~~3a. Customer's outstanding debts are above the threshold:~~ …

Postconditions:

```
System has initiated an order for the customer, has documented
payment information with price reduction, and has registered the
order with the customer.
```

Rule.5 No Constraints for Use Case Preconditions

The Design-by-Contract principle in the OO domain [25] and the LSP [21] also require a subtype to either maintain or weaken the preconditions of a parent operation. As we know from Section 3 this is mainly for ensuring behavioural conformity when substituting child objects for parent objects. However, we have seen that for use case performances there neither is substitutability (see Section 4.1), nor does Design-by-

Contract apply (see Section 1.2); further use case preconditions are checked by the system, not by the triggering actor instance (see Section 1.2). Consequently, there is no need for use cases to enforce the same behavioural conformity semantics as desired for objects.

Please note however that the proposed Rules 3 to 5 still enable, but do not *necessarily enforce*, strict behavioural subtyping.

6 Critical Closing Remarks

6.1 UCI vs. Include/Extend Revisited

From the OO domain we know that any inheritance can alternatively be expressed by object aggregations, i.e. by whole/part relationships, and delegation (see Section 3). Therefore, a use case model employing Rule 4 can also be expressed via Include-relationships as shown in Fig. 7.

Fig. 8 shows the Include-relationships version of Fig. 6 (in Example 4, above).

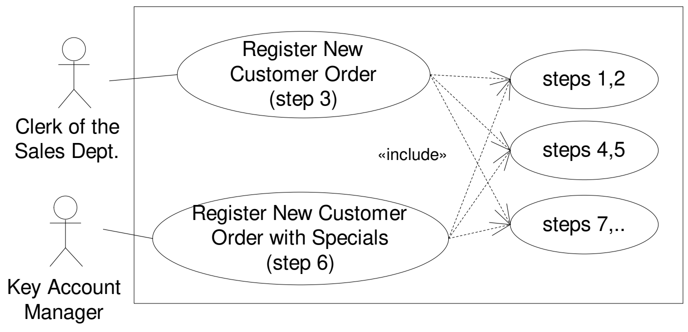
Clerk of the Sales Dept.
Key Account Manager
Register New Customer Order (step 3)
Register New Customer Order with Specials (step 6)
«include»
steps 1,2
steps 4,5
steps 7,..

Fig. 7: Example 4 realised with Include. Numbers

Even though that it is possible to solve a problem with Include where UCI appears appropriate it is obvious that an Include solution entails a greater graphical and textual complexity. This impacts on the reader's convenience and reading efficiency: in Fig.6 only one document (for "RegisterNewCustomerOrderWithSpecials") needs to be opened while in Fig.7 it would be four documents ("RegisterNewCustomerOrder-WithSpecials" and three inclusion use cases), or at least 4 different document sections have to be looked up. Since size of use case models can be an issue in practice, UCI should contribute to keeping the use case model size at a minimum, and, therefore, ease reading and reviewing.

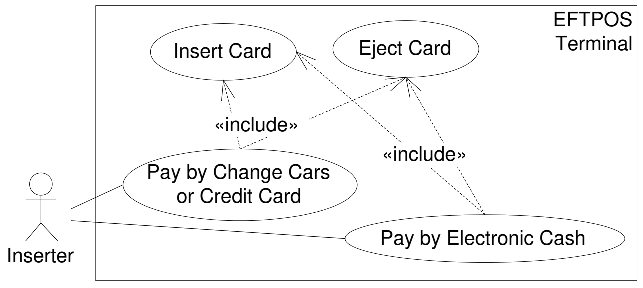

Fig. 8: Example 3 realised with Include

In any case, what cannot be expressed which Include or Extend is the idea of generic identifiers and generic interaction steps: neither Extend nor Include is an alternative in situations as shown in Examples 2 and 3 as these relationships do not support parameters and are not capable of *propagating* any type of “abstractness” [26]. Therefore, Rules 1 and 2 serve as further clear semantical distinction criteria.

6.2 Comprehensibility of UCI

One might argue that UCI only appears simple and understandable to software engineers, UML modellers and programmers since, historically, this audience has been mostly familiar with inheritance. Unfortunately, this audience is not the only one that deals with use case modelling: the use case calculus encompasses a requirements elicitation, modelling, and textual documentation technique and, thus, by definition, does require non-IT business domain experts to be involved. In my industry career as a requirements engineer I have experienced that the speaking in terms of *substituting placeholders*, *rewriting and adding interaction sequences*, and *appending more to goals and use case results* can be understood (and is often found helpful) by such stakeholders, in contrast to OO-related terminology like *Open-Closed*, *polymorphism*, *abstract classes*, or *subtyping* etc. However, UCI still remains a hard to understand concept in practice, irrespective of the added values identified and guidance provided by my solution proposals above. Without effective training, operational coaching, and industrial experience the benefits of UCI will not necessarily show.

References

1. Adolph S., Bramble P., Cockburn A., Pols A. “Patterns for Effective Use Cases”, Addison-Wesley, 2003
2. Ambler S. “Reuse in Use-Case Models: Extend, Include, and Inheritance”, Agile Modeling Essay, http://www.agilemodeling.com/essays/useCaseReuse.htm#InheritanceUC
3. Arlow J., Neustadt I. “UML2 and the Unified Process Second Edition”, Addison-Wesley, 2005
4. Armour F., Miller G. "Advanced Use Case Modeling", Addison-Wesley, 2001
5. Bittner K., Spence I. “Use Case Modelling”, Addison-Wesley, 2003
6. Booch G., Jacobson we., Rumbaugh J. “Rational Unified Process”, Rational Software Corporation
7. Bracha G., Cook W. “Mixin-Based Inheritance”, OOPSLA/ECOOP ’90, ACM SIGPLAN Not. 25
8. Cardelli L., Wegner P. “On Understanding Types, Data Abstraction, and Polymorphism”, Computing Surveys, December, 1985
9. Chambers C., Ungar D., Chang B.-W., Holzle U. ”Parents are Shared Parts of Objects: Inheritance and Encapsulation in Self”, 1999, Lisp Symbolic Computing
10. Cockburn A. "Writing Effective Use Cases", Addison-Wesley, 2001
11. Cook S., Daniels J. “Designing Object Systems – Object-Oriented Modelling with Syntropy”, 1994
12. Dijkstra, E.W. “On the role of scientific thought”, 1974, in “Selected Writings on Computing: A Personal Perspective”, Springer, 1982, p.60-66
13. Fowler M., Cockburn A., Jacobson I., Anderson B., Graham I. "Question time! About use cases", Procs. 13th OOPSLA, pub. ACM Sigplan Notices 33 (10)
14. Genilloud, Frank "Use Case Concepts from an RMODP Perspective", JOT, vol. 4, no. 6, Special Issue: Use Case Modeling at UML-2004, Aug 2005
15. Génova G., Lloréns J., Quintana V. “Digging into Use Case Relationships”, UML 2002 Conference, LNCS 2460, Springer Verlag, Germany, 2002
16. Hitz M., Kappel G. "UML @ Work", dpunkt Verlag, Germany, 1999
17. Jacobson I. “ Use Cases - Yesterday, Today, and Tomorrow”, Rational Software Corporation,

18. Jacobson I. Christerson M., Jonsson P., Övergaard G. “Object Oriented Software Engineering – A Use Case Driven Approach”, Addison-Wesley, 1992
19. Jacobson I. “The Road to the Unified Software Development Process”, Cambridge University Press, SIGS Reference Series, 2000
20. Kulak D., Guiney E. “Use Cases – Requirements In Context”, Addison-Wesley, ACM Press, 2000
21. Liskov B. H., Wing J. M. "Behavioral Subtyping Using Invariants and Constraints", School of Computer Science, Carnegie Mellon University, Pittsburgh, July, 1999
22. Metz P., O’Brien J., Weber W. “Against Use Case Interleaving”, UML 2001 conference, LCNS 2185, Springer Verlag, Germany, 2001
23. Metz P., O’Brien J., Weber W. “Specifying Use Case Interaction: Types of Alternative Courses“, in Journal of Object Technology (JOT), Issue March/April 2003,
24. Metz P. “Revising and Unifying the Use Case Textual and Graphical Worlds”, PhD thesis, Cork Institute of Technology, 2004
25. Meyer B. “Object-Oriented Software Construction”, 2nd Edition, Prentice Hall, 1997
26. OMG Unified Modeling Specification, v1.4, Sept. 2001,OMG, UML v2.0 Superstructure 03-08-02
27. Övergaard G, Palmkvist K. "A Formal Approach to Use Cases and Their Relationships", Proceedings of "UML'98: Beyond the Notation", LCNS 1618
28. Övergaard G, Palmkvist K. “Use Cases: Patterns and Blueprints”, Addison-Wesley, 2004
29. Rosenberg D., Scott K. “Use Case Driven Object Modeling with UML”, Addison-Wesley, 1999
30. Schneider G., Winters J. “Applying Use Cases – A Practical Guide”, Addison-Wesley, 1997
31. Sendall S. “Specifying Reactive System Behavior”, Ph.D. thesis, École Polytechnique Fédérale de Lausanne, 2003
32. Simons A. "Use Cases Considered Harmful", Procs. TOOLS-29 Europe, 1999
33. Stein L. “Delegation is Inheritance”, OOPSLA ’87 Conference Proceedings, ACM Sigplan Not.22, 12 (Dec.)
34. Strachey C. “Fundamental Concepts of Programming Languages”, 1967
35. Taivalsaari A. “On the Notion of Inheritance”, ACM Computing Surveys, Vol. 28, Sept. 1996
36. Use Case Workshop “Industrial Use Case Modeling”, 7th UML 2004 Conference, October, 2004
37. Van den Berg K., Simons A. "Control-Flow Semantics of Use Cases in UML", Information and Software Technology, Elsevier Science B.V., 19