The Role of "Roles" in Use Case Diagrams

Alain Wegmann[1], Guy Genilloud[1]

[1] Institute for computer Communication and Application (ICA)
Swiss Federal Institute of Technology (EPFL)
CH-1015 Lausanne, Switzerland
icawww.epfl.ch
{alain.wegmann, guy.genilloud}@epfl.ch

Abstract: Use cases are the modeling technique of UML for formalizing the functional requirements placed on systems. This technique has limitations in modeling the context of a system, in relating systems involved in a same business process, in reusing use cases, and in specifying various constraints such as execution constraints between use case occurrences. These limitations can be overcome to some extent by the realization of multiple diagrams of various types, but with unclear relationships between them. Thus, the specification activity becomes complex and error prone. In this paper, we show how to overcome the limitations of use cases by making the roles of actors explicit. Interestingly, our contributions not only make UML a more expressive specification language, they also make it simpler to use and more consistent.

1 Introduction

The Unified Modeling Language (UML), standardized by the Object Management Group (OMG) in 1996, aims at integrating the concepts and notations used in the most important software engineering methods. UML is today widely used by the software development community at large. While the bulk of the integration of the concepts is completed, there are still improvements to be made in their consistency. Such improvements could increase the expressive power of UML while reducing its complexity.

System design frequently starts with business modeling, i.e. modeling the context of the system to be developed. The aim is to understand the processes in which the system participates and the system's functionality. UML proposes the *use case model* to describe the system's functionality.

Ivar Jacobson initially defined use case models in [p. 127, 7]: *"The use case model uses actors and use cases. These concepts are simply an aid to defining what exists outside the system (actors) and what should be performed by the system (use cases)"*. According to this description, a use case represents a part of the system's functionality. UML defines use cases in a similar manner in [p. B-19, 12]: "*the specification of a*

sequence of actions, including variants, that a system (or other entity) can perform, interacting with actors of the system..."

Use case models are similar to role models because their intent is to capture the roles of each participant in an action. Role models are defined by Trygve Reenskaug in the OOram method [14]. The method aims at achieving a design by an understanding of how roles collaborate to achieve a goal (i.e. defining a "role model"). Roles are then implemented by programming language objects. For our discussion, the OOram important elements are: (1) roles help with the separation of concerns; even if an object can fulfill more than one role, the designer can still analyze each role individually; (2) roles focus on the notion of responsibilities (i.e. messages accepted and those sent by a role), as opposed to classes that focus on capabilities (i.e. putting more emphasis on the message accepted as opposed to those sent by a class). This method influenced significantly UML and, in particular, the interaction diagrams (i.e. collaboration and sequence diagrams). A good overview of the importance of role models can be found in [11].

Use case models can be used to model functionality of entities at different levels of abstraction: for example business entities (e.g. companies) [8], sub-systems (e.g. existing IT systems to be integrated), components or even programming language classes. In this paper we are particularly concerned with the refinement from business models to system specification models. In the business model, the system of interest is the enterprise and the actors are the people, companies or IT systems interacting with the enterprise. In the system specification model, the system of interest is usually an IT system, which needs either to be developed or modified and the actors are the entities in direct contact with the system of interest. From our experience in consulting, we raised several modeling questions about the utilization of use case diagrams that document system specification models.

The modeling questions we identified are related to the representation of the systems in UML diagrams, to the impossibility of specifying some important requirements of use cases, and to the reuse of use cases. Ian Graham mentions already some of these problems in [4].

Desmond D'Souza and Allan Wills provide a partial answer with their Catalysis method [1]. In their method, they first analyze the role of an IT application in its business environment, and define the system specification independently from the implementation details. They then implement the system by defining a collaboration of "pluggable-parts" such as programming language classes or components. They define collaboration as a "*set of related actions between typed objects playing defined roles in collaboration*" [p. 716, 1]. In Catalysis, the use case is a means to specify a collective behavior of entities without specifying the individual behavior of each entity. This idea came originally from DisCo [9].

Our paper proposes to extend the Catalysis definition of use cases by leveraging on the concept of role. Our propositions allow for the improvement of the use case expressiveness and should lead to a simplification of UML.

The plan of this paper is: Section 2: identification of modeling questions related to use cases, Section 3: discussions of the questions and proposition of extensions to the use case modeling technique, Section 4: propositions of modifications to UML, Section 5: case study revisited using the extended use case modeling technique, Section 6: future work.

2 Modeling Questions

In this section, we present five modeling issues related to use cases. We illustrate these issues with an example of Company, a chain store. The Company has one Corporate HQ (headquarter) and several Stores (see Fig. 1).

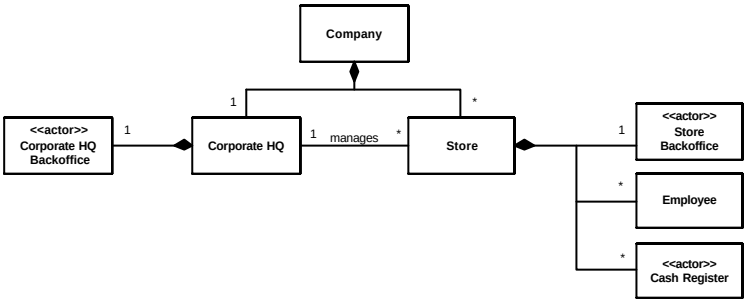

Fig. 1: Class diagram describing Company structure

The Company began an IT project to automate the Cash Registers of its Stores. The functionality to be provided is (see Fig. 2): "sell Goods" (i.e. the Cashier computes price to be paid by the Customer and then proceed with the payment), "till Balance" (i.e. the Cashier and the Manager check the content of the cash drawer) and "transfer Price" (i.e. new price lists are transferred from the Corporate HQ to all Cash Registers with the collaboration of the Store Backoffice).

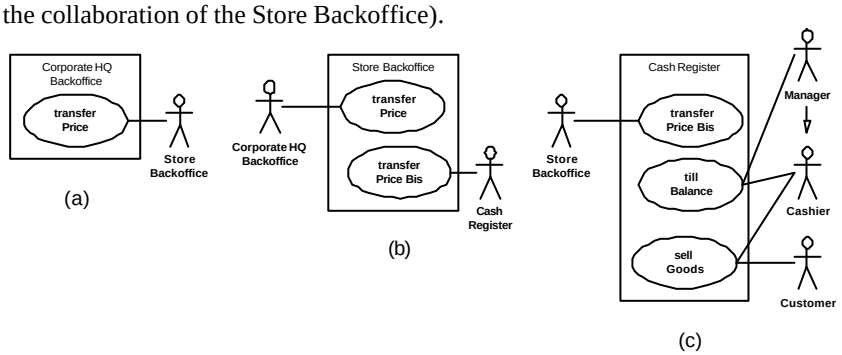

Fig. 2: (a) Corporate HQ Backoffice use case diagram, (b) Store Backoffice use case diagram, (c) Cash Register use case diagram

This example raises the following points:

1. As the “transfer Price” business process specification involves three types of IT systems (Corporate HQ, Store Backoffice and Cash Register), we must have three separate use case diagrams (one per system type). We would much rather have one diagram representing all system types to better understand the role of each system type relative to the other system types. We could use an interaction diagram. We would not necessarily reject this solution if in UML, as in Catalysis, interaction diagrams could represent use case occurrences (in a similar manner as stimuli are represented). Without the ability to represent use case occurrences, we would have to refine the interactions between the systems down to the level of stimuli exchanges. That is, we would be forced to provide too many details for what is needed. This raises Question 1: “How can we model, in one use case diagram, a business process specification between multiple system types and actor types?”
2. As represented in Fig. 2b, the Store Backoffice system will perform occurrences of two use cases: “transfer Price” and “transfer Price bis”. These two use case specifications are identical, except for, in each occurrence, the actors are different and the system plays a different role (sender in one case and receiver in the other case). This forces the designer to have two independent use case specifications (“transfer Price” and “transfer Price bis”). Of course, we want to have just one use case specification “transfer Price”. This raises Question 2: “How can a system play different roles in different occurrences of a same use case specification?”
3. Traditionally use case diagrams do not express multiplicities. In our example, this prevents the modeler from specifying if the “transfer Price“ use case involves only one recipient (unicast) or many (multicast). This raises Question 3: “How can we capture constraints on the number of actor instances in a use case occurrence?”
4. When the prices are transferred, “transfer Price” should occur first, followed by “transfer Price bis”. UML use case diagrams alone do not provide a way to specify such relationships between use cases. As a result the semantics of use case diagrams are often unclear. This raises Question 4: “How could we represent constraints on when use cases may occur?”
5. The concept Store Backoffice is shown as an actor (Fig. 2c) or as a system (Fig. 2b) in the use case diagrams and as a class, possibly stereotyped with <<actor>>, in the class diagram (Fig. 1). The same concept is shown with a different diagram element, so what is specific to actors? This raises Question 5: “ What is an actor?”

3 Extension to Use Case Modeling Technique

In this section, we will analyze the questions of Section 2 and propose possible solutions.

To be precise, this paper will use the RM-ODP definition [Section 9, 6] of the terms *type*, *class*, *specification*[1], *instance* (used for concepts such as objects, components, etc.) or *occurrence* (used for concepts such as messages, actions, etc). The use of these terms is illustrated in the following example: an actor specification defines the features of an actor, an actor instance defines an actual actor entity, an actor class defines a set of actors that share common characteristics, and an actor type defines the common characteristics of the actors belonging to the class.

3.1 Representation of the System

In this section, we answer Question 1: "How can we model, in one use case diagram, a business process specification between multiple system types and actor types?"

To model a business process that involves multiple system types and actor types, we need to be able (1) to indicate which system type realizes which use case, and (2) to model the use cases that do not directly involve systems (i.e. those use cases that are only between actors). Currently UML use case diagrams force the designer to have only one system of interest in a use case diagram by either representing only one system (drawn as a box around the use cases) or none at all. This excludes from the diagram, the use cases not involving directly the system of interest.

A possible answer can be found in Catalysis [1], a method that defines use cases as not system-centric. Their definition of use case is "*a joint action with multiple participant objects that represent a meaningful business task, usually written in a structured narrative style. Like any joint action, a use case can be refined into a finer-grained sequence of actions*" [p. 722, 1]. A joint action is defined as: "*a change in the state of some number of participant objects without stating how it happens and without yet attributing the responsibility for any of it to any one of the participants*" [p. 158, 1]. A use case may be described on two levels. First level is a declarative description, defined as the change of state of all use case participants resulting from its execution. The declarative description is composed of pre- and post-conditions. It puts an emphasis on the collective behavior of all participants. Second level is an operational description defined as a refinement of the declarative description in which the joint action is decomposed into smaller grain actions. These actions are either joint actions or localized actions. A localized action is defined as "*a one-sided specification of an action focused entirely on a single object and how it responds to a request,*

[1] To have a terminology closer to UML, we define *specification* as a synonym for the RM-ODP term *template*. We also consider *occurrence* as a synonym for the RM-ODP term *instance*.

without regard to the initiator of that request" [p. 715, 1]. The operational description puts an emphasis on the individual behavior of each participant.

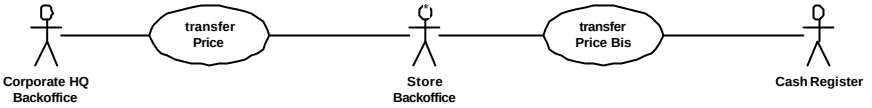

Fig. 3: Use case diagram representing systems with actors (UML requires that actors are used as participants to use cases)

The Catalysis definition of the use case does not make any reference to "the system". Use cases are therefore no longer system centric[2]. The implication is that three use case diagrams shown in Fig. 2 may now be represented in one diagram, as shown in Fig. 3.

3.2 Reuse of Use Case Specifications

We answer Question 2: "How can a system play different roles in different occurrences of a same use case specification?"

The Catalysis use case definition does not answer this question. Catalysis, as well as UML, forces the designer to have one use case specification for each group of actors involved (see "transfer Price bis" use case in Fig. 3). A possible answer with Catalysis is to use a collaboration framework [p. 346, 1] to show that two use cases with different names are of the same type. However this does not solve the problem, as a same use case specification cannot be used by two different groups of actors.

Use case specifications explicitly refer to actors and this is the source of the reuse problem. Introducing roles instead of actors solves it. This is consistent with the UML definition of actors as a set of roles.

UML defines role as: *"the named specific behavior of an entity participating in a particular context. A role may be static (e.g., an association end) or dynamic (e.g., a collaboration role).*" We focus on the first part of the definition, the second part is not important for the present discussion. By using this definition and replacing entity by actor and context by use case we can show that an actor may be identified by its role in the use case context (rather than by its name). Thus, roles provide the mechanism needed for making use case specifications independent of actors. Use case specifications may then be reused between different groups of actors and can also refer to a same actor instance playing different roles in different occurrences. Of course, we must have a mechanism to bind roles to actors. This can be done in use case diagrams by writing the role as the associationEndRole on the association between the use case and the actor (see Fig. 4).

[2] We will propose a use case definition for UML in Section 4.0

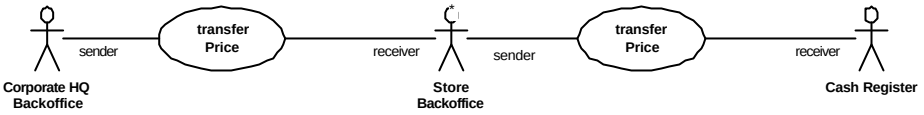

Fig. 4: Example of use case specification reuse in one use case diagram

Some readers may be puzzled to see two use cases with the same name involving two different groups of actors. This is not uncommon as it is analogous to having two associations with the same name but between different classes in a class diagram. We still need to understand what is meant by two "transfer Price" use cases in a same diagram. For example, in Fig. 4, the use case on the left corresponds to the class of use cases that are instantiations of the "transfer Price" use case specification with its role sender referring to Corporate HQ Backoffice and the role receiver referring to the Store Backoffice.

Our approach to explicitly represent roles enables reusing a same use case specification between different groups of actors in a same use case diagram or in different diagrams. In addition, it is consistent with UML and in particular with: (1) the definition of role, (2) the meta-model (use cases and actors are classifiers with an association between them), (3) the notation of roles in class diagrams (in which the roles are represented at the association end).

3.3 Constraints about Number of Instances Participating in a Use Case

We address Question 3: "How can we capture issues related to number of actor instances in a use case occurrence?".

It is not clear whether the use case modeling technique has provisions for representing the number of actor instances (of the same actor type) participating in a use case occurrence. The UML notation guide shows a few examples of multiplicities [p. 3-93, 12]. However, the meta-model does not acknowledge the existence of an actor instance and it is not clear if role is a type or an instance.

The difference between type and instance is often unclear as illustrated in the following two examples "*roles (in collaborations) are somewhat between types and instances*" [p. 3-15, 12] and "*if there can be more than one instance corresponding to a given ClassifierRole, one of these instances is selected to represent them all*" [13]. We believe that the difficulty in deciding if something is a type or an instance is based on the fact that people tend to think in terms of prototypes (i.e. an instance of a type). This is thoroughly discussed by George Lakeoff in [10]. The prototype defines a type by using a specific instance especially representative of the type. But, at the same time, the prototype denotes one or more actual instances. For example an instance of a policeman in uniform is considered as defining a type (i.e. the predicate that allows to decide whether a man is a policeman) but is an instance at the same time (i.e. the man currently in the middle of the crossing). This mechanism of prototype explains why,

sometimes, concepts are difficult to categorize as type or instance. Unfortunately, the prototype mechanism is not applicable in UML. Types have to be defined explicitly. For this reason, *type models* (e.g. class diagrams) and *instance models* (e.g. object diagrams) have to be developed. Based on this, we state:

1. All concepts exist as instances (at a specific location in time and space).
2. All concepts may be categorized into classes by means of types (i.e. predicates).
3. Instances are useful for considering interactions between concepts
4. Classes are useful for working with instances in the sense that we do not need to look at each instance separately.

We propose that UML defines all concepts as both a concept type and a concept instance. For most concepts, this duality type / instance already exists. The terms chosen to denote the types and the instances are usually quite different from each other, for example: message and stimulus, object and class, use case [class] and use case [instance]…. Our proposition is to add definitions for actor [instance] and role [instance] to the UML meta-model. Acknowledging the existence of actor instances in UML is consistent with the possibility to express multiplicities in use case diagrams (as multiplicities express constraints on the number of instances). This is illustrated in Fig. 5.

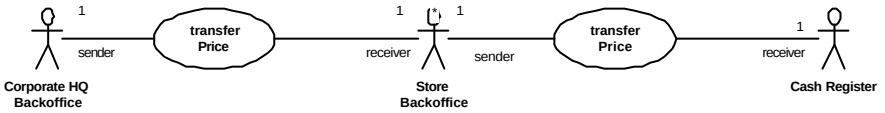

Fig. 5: Example of multiplicity in use case diagram

Fig. 5 illustrates the use of multiplicities in use case diagrams. Considering the "transfer Price" use case, writing a multiplicity of 1 on the receiver role indicates that a use case corresponds to a unicast. On the other hand, writing a multiplicity of 1..* would indicate a multicast of price information.

The multiplicity notation is analog to the one used in class diagrams. The multiplicity is on the actor side of the association and expresses constraints on the number of instances involved in one use case occurrence. We purposely omit multiplicity on the use case's side of the association, as an actor may almost always participate in an unlimited number of occurrences of use case.

Our approach is to systematically define all concepts as types and instances. This allows multiplicities to be represented in type models (e.g. class diagram or use case diagram) as multiplicities represent constraints on the number of instances.

3.4 Constraints on Use Case Occurrences

We analyze Question 4: "How could we represent constraints on when use cases may occur?"

Actor instance and use case occurrence concepts enables the drawing of use case instance diagrams as shown in Fig. 6. By numbering the occurrences of use cases, it is then possible to illustrate the sequence in which use cases will be executed. The sequence-numbering notation is the same as the one defined in collaboration diagrams. Its limitations are also the same. Further work needs to be done on specifying execution constraints beyond what is already defined in interaction diagrams (e.g. *"constraints may include for example sequentiality, non-determinism, concurrency or real-time constraints*" [Section 8, 6]).

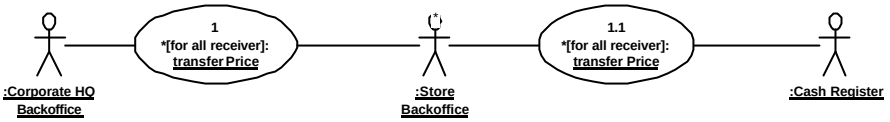

Fig. 6: Example of use case instance diagram

Note that Pavel Hruby proposes to use state diagrams to specify execution constraint between use cases [5]. His approach is complementary to our proposal.

3.5 The Role of Actors

Question 5: "What is an actor?" is now addressed.

UML defines actors as *"a coherent set of roles that users of use cases play when interacting with these use cases. An actor has one role for each use case with which it communicates*". As all entities realize a set of roles, it is not clear what is so unique about actors?

Catalysis defines actors as *"external roles participating in an action*" [p. 592, 1]. In general, they represent the use case participants with diagram elements corresponding to the actual entity (e.g. actor, system, component, programming language class, etc.). Unfortunately UML specifies that participants in use cases are actors (and not any other possible entities such as sub-system, components, programming language classes). Should this restriction be lifted?

To understand the specificity of actors and whether entities other than actors can be represented in use case diagrams, we need to consider how actors are used:

1. Traditionally actors represent entities exterior to the system of interest. For example, in Fig. 2c, the Customer actor represents a person coming in the Store to purchase goods.
2. An actor links use cases together by performing a number of roles. For example, in Fig. 5, the Store Backoffice actor receives the prices by participating in a class of "transfer Price" use cases and then sends these prices by participating in a second class of "transfer Price" use cases.
3. An actor represents, in a use case diagram, an entity coming from another diagram (or vice-versa). Using the same name for an actor and an entity in another diagram

establishes this relation. For example, in Fig. 5, the Store Backoffice actor represents the Store Backoffice shown in the class diagram in Fig. 1.

4. An actor is sometimes used as a means to indicate explicitly which entity realizes a set of roles. This is done either by using a <<realize>> relationship between an actor (representing the roles) and an entity (realizing the roles) diagram elements, or by adding "/name" to an entity identifier to represent its role (where "name" is the name of an actor).
5. An actor may have a generalization relationship with another actor [p. 3-92, 12]. For example, in Fig. 2c, Manager is a generalization of Cashier. That is, all Managers are also Cashiers. Or, in other words, the Manager actor type is a subtype of Cashier in the sense that a Manager can perform all the roles of a Cashier.

The first use illustrates the specificity of the actor concept compared to the other entity concepts. An actor is used when the designer needs to model only a part of the behavior of an entity (which is typically the case for entities external to the systems of interest as the designer does not have to consider or to specify their full behavior).

The second use does not require a specific actor concept. All entities may realize multiple roles; so all entities may be used in use case diagram for linking two use cases together. Only the definition of the use case diagram forces the systematic use of actors.

The third use is quite artificial and is a direct consequence of the use case diagram definition that allows for the representation of actors only, use cases and possibly one system. If the actual entities could be represented in the use case diagram (with their original diagram element as done in Catalysis), the use case diagram would gain in clarity as the designer could decide to represent the actual entity fulfilling roles rather than using an indirection via an actor.

The fourth use becomes marginal if use case diagram elements can represent any entities as participants in the use cases. Actors can still be used when the designer does not want to specify which entity will realize the role (e.g. definition of a collaboration framework involving multiple use cases). In such cases, an actor represents a composite role (called "actor role") played in a specific context (called "actor context"). The actor context is the set of use cases in which the actor participates. The role played by one actor in one specific use case is called "use case role". The actor role is the composition of all the use case roles. Based on this, we recommend defining actor as a composition of roles (as opposed to a set of roles, which is not a role). Note: when needed, the designer explicitly states which entity realizes the actor role. In such cases, an entity plays a composite role (called "entity role") in a specific context (called "entity context"). The entity role is the composition of all actor roles the entity realizes. The entity context is the composition of the corresponding actor contexts.

The fifth use of actors is to show generalization relationships between participants in use cases. In our example, the generalization relationship between Manager and Cashier shows that a Manager can perform the roles of a Cashier. It is not intended to signify that the role of the Cashier in "sell Goods" is a Manager's role (only that it is a role that a manager can realize). It would be preferable to express that a same Employee could realize the Manager's roles and the Cashier's roles (by making explicit who real-

izes the two sets of roles) rather than to merge these two sets of roles into one (by using the generalization relationship). This allows the designer to keep both set of roles separate. The generalization between actors should be carefully used. In general, the <<realize>> relationship is more appropriate for assigning a set of roles to an entity.

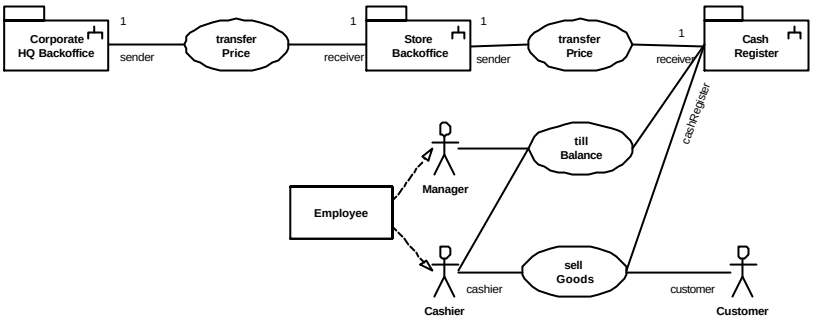

Fig. 7: Example of use case diagram in which participants are subsystems and actors.

Fig. 7 illustrates our recommendation for the representation of any entity diagram elements in a use case diagram and to redefine Actor (see Section 4.0). CorporationHQ Backoffice, Store Backoffice, and Cash Register are represented using the diagram element corresponding to the actual entity (a subsystem to represent an IT system). Customer remains an actor, as it is an entity that will remain partially specified (as an external participant, only its role in the context of "Sell Goods" is interesting to us). Manager and Cashier remain actors, as they represent roles, which will have to be mapped to an actual entity. This mapping is made explicit by using the two <<realize>> relationships. Even if the resulting diagram appears more complex than a diagram using only actors (more type of entities are represented), it is actually simpler to use (as it removes unnecessary indirection between diagram elements).

4 Modifications to UML

In this section we discuss the impact of our proposal on UML. We propose the following definitions:

Use case [class[3]] - the specification of the change of state of a group of entities willing to achieve some purpose. This change of state can be described either as the result of the occurrence of one abstract action involving all the entities or as the result of the occurrences of sequences of individual actions involving individual entities.

Use case [instance] – an occurrence of a use case [class].

[3] Class is used here with its UML meaning, i.e. specification (or ODP template).

Actor [class] –the specification of a role defined as the composition of the roles that a participant of use cases play when interacting with these use cases..
Actor [instance] - an instance of an actor [class].

By introducing the above definitions in UML and further by relaxing the constraints on the diagram elements allowed in use case diagrams, we address most of the raised modeling questions. This implies that current UML case tools need only minimal changes to apply our extended modeling technique.

To express execution constraints on use cases, the meta-model needs to be extended to incorporate the missing concepts of: Actor Instance, Subsystem Instance, and Instance Role[4].

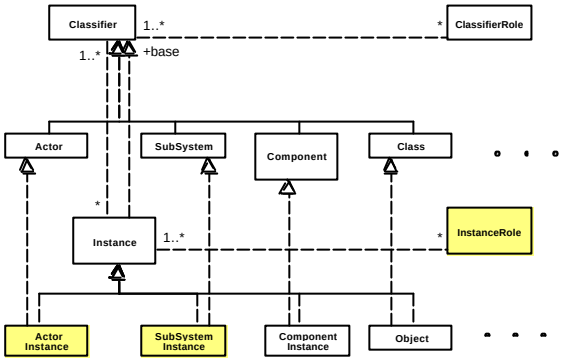

Fig. 8: Elements of meta-model related to the classifier – instance relationship

The proposed modifications to the meta-model are illustrated in Fig. 8. They make it more consistent as they remove some exceptions (Classifier concepts without corresponding Instance concepts).

5 Application of our Suggestions

In Section 2, we presented "classic" use case models (Fig. 1 and Fig. 2). Fig. 7 and Fig. 9 present analog models that reflect the use of our new definitions of use case and actor. Note the consistency between the class diagram (Fig. 9) and the corresponding use case diagram (Fig. 7), which is not the case in the classic models (Fig. 1 and Fig. 2).

[4] We name the meta-class InstanceRole rather than RoleInstance to be consistent with ClassifierRole.

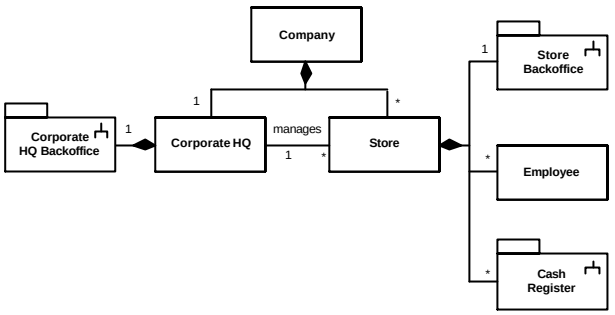

Fig. 9: Class diagram describing Company organization

Our new use case modeling technique can be compatible with the standard UML technique by:

1. Allowing optional representation of use case roles (in Fig. 7 the use case roles in “till Balance” are not specified).
2. Allowing the use of a rectangle around a set of use cases to represent that an actor participates in all use cases represented in the rectangle. Note that a use case diagram might have more than one of these rectangles.

Sometimes, roles are difficult to name. As the roles are bound to the use case, it is possible to use the same name to denote an actor and its roles in the use cases. This name might start with a capital letter when denoting an actor and a lower case when denoting the role. In our example in Fig. 7, Cashier has the “cashier” role in the “sell Goods” use case. This convention is consistent with the one used in class diagrams to denote AssociationEndRole.

6 Future Work

In this paper, we proposed new definitions for actor and use case, as well as the addition of new classes to the UML meta-model. An advantage with these changes is that all entities may be represented in all UML diagrams. The notational techniques, that we propose for use case and use case instance diagrams, are similar from those of class diagrams and interaction diagrams. We believe that this is more than a mere coincidence: the essence of these diagrams is the same, however they differ by their notational techniques. We believe that they can be integrated or unified. Further work needs to be done towards this integration. The results would simplify UML further and would lead to the following benefits: (1) simpler utilization, (2) better specification capabilities, and (3) simplification of case tools.

Conclusion

Use cases are the modeling technique of UML for formalizing the functional requirements placed on systems. In this paper, we have shown several quite important limitations of this technique. It is not possible to model the context of a system beyond its immediate environment (e.g., if two actors exchange information related to their use of a system, this communication cannot be shown in a use case diagram). Likewise, it is impossible to show how several systems are related, even though those systems support a same business process. Reuse opportunities for use case specifications are denied, because use case specifications are directly tied to their associated actors. And execution constraints between use case occurrences cannot be shown or specified in any way.

These limitations can be overcome to some extent by the realization of multiple models and multiple diagrams of various types. But the more diagrams and models there are, the larger the amount of work to be done, and there is the problem of specifying and maintaining the relationships between all these models and diagrams. In this paper, we showed that another approach was possible and quite effective.

This approach relies on three principles: making the roles of use case participants explicit, representing use case participants with their actual diagram elements, and treating the system as any other use case participants. These three principles would require very limited changes to UML: the definitions of actor and use case must be revised.

A complementary idea is to enable modeling at the level of use case occurrences and actor instances (the diagrammatic techniques are borrowed from those of interaction diagrams). We think that modeling at this level is invaluable for relating use cases together and for expressing execution constraints between them. The necessary changes to UML are again quite modest. The meta-model needs to be extended to incorporate the missing concepts of: Actor Instance, Subsystem Instance, and Instance Role.

Quite importantly, all the modifications we propose for UML increases its consistency. As a result, they not only contribute to make UML a more expressive specification language, they also make it a simpler language to understand and use.

Acknowledgments

John Donaldson (Compaq Professional Services, Geneva, Switzerland) and Frederic Bouchet (Nortel Professional Networks, Bussigny, Switzerland) helped to identify the problems of modeling the reengineering of a business process at the abstraction level of IT systems. Special thanks to an anonymous reviewer for providing us with numerous useful suggestions.

References

1. D'Souza Desmond F., Wills Alan Cameron: Objects, Components, and Frameworks with UML – The Catalysis Approach. Addison-Wesley (1999) (ISBN 0-201-31012-0).
2. Genilloud Guy, Wegmann Alain: On Types, Instances, and Classes in UML. European Conference - Object-oriented Programming (ECOOP), Sophia-Antipolis, France (2000) (http://icawww.epfl.ch).
3. Genilloud Guy, Wegmann Alain: A Foundation for the Concept of Role in Object Modelling. Enterprise Distributed Object Computing Conference (EDOC), Makuhari, Japan. (2000) (http://icawww.epfl.ch).
4. Graham Ian: Requirements Engineering and Rapid Development: an Object-oriented approach. Addison-Wesley (1998) (ISBN 0-201-36047-0).
5. Hruby Pavel: Structuring Specification of Business Systems with UML (with an Emphasis on Workflow Management Systems). OOPSLA Workshop: Business Object Design and Implementation IV: From Business Objects to Complex Adaptive Systems, Vancouver B.C. (1998).
6. ISO/IEC ITU-T: Open Distributed Processing – Basic Reference Model – Part 2: Foundations. Standard 10746-2, Recommendation X.902 (1995) (http://isotc.iso.ch /livelink/livelink/fetch/2000/2489/Ittf_Home/PubliclyAvailableStandards.htm).
7. Jacobson Ivar, Christerson Magnus, Jonsson Patrick, Övergaard Gunnar: Object-Oriented Software Engineering. Addison-Wesley (1992) (ISBN 0-201-54435-0).
8. Jacobson Ivar, Ericsson Maria, Jacobson Agneta: The Object Advantage – Business Process Reengineering with Object Technology. Addison-Addison Wesley (1995) (ISBN 0-201-42289-1).
9. Kellomäki Pertti, Mikkonen Tommi: Design Templates for Collective Behavior. European Conference – Object-oriented Programming (ECOOP), Sophia-Antipolis, France (2000) 277 – 295.
10. Lakoff George: Women, Fire and Dangerous Things – What Categories Reveal about the Mind. Chicago Press (1987) (ISBN 0-226-46804-6).
11. Li Qing, Wong Raymond: Multifaceted object modeling with roles: A comprehensive approach. In: Information Sciences 117, Springer-Verlag, (1999) 243-266.
12. OMG: Unified Modeling Language Specification, Version 1.3 (1999) (www.omg.org).
13. Reenskaug Trygve: UML Collaboration and OOram semantics – New version of green paper. 2nd ed, Nov. 8, 1999 (http://www.ifi.uio.no/~trygver/documents).
14. Reenskaug Trygve, Wold Per, Lehne Odd Arild, Working With Objects: The OOram Software Engineering Method. Manning Publications (1996) (ISBN 0-13-452930-8).