The Role of "Roles" in Use Case Diagrams

Alain Wegmann, Guy Genilloud

May 2000

Technical Report DSC/2000/024

EPFL-DSC
CH-1015 Lausanne *http://dscwww.epfl.ch*

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

Traditionally, software designers had to live with different concept definitions and notations, depending on the development method used. This was hindering the software community by increasing the communication barrier between teams and distracting the developers from the important issues. To address these issues, OMG (Object Management Group) standardized in 1996 the Unified Modeling Language. UML is based on the integration of concepts coming from the most important software engineering methods. Since then, UML has become 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 [5]: "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 functionality of the system. UML

defines use cases in a similar manner [9]: “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 were initially designed for modeling the functionality of IT systems, but they can also be used for modeling the functionality of entities at different levels of abstraction such as: business entities (e.g. companies) [6], sub-systems (e.g. existing IT systems to be integrated), components or even classes.

Use case models can be related to role models because their intent is to capture the roles of each participant to an action. To understand how roles are connected to use case models, it useful to look at *role modeling* as defined by Trygve Reenskaug in the OOram method [11]. This method influenced significantly UML and, in particular, the interaction diagrams (i.e. collaboration and sequence diagrams). In OOram, a role model is defined as: “*an abstraction on the object model where we recognize a pattern of objects and describe it as a corresponding pattern of roles*”. The elements of OOram important for our discussion are: (1) roles help with the separation of concerns (i.e. an object can fulfill more than one role; a role modeling approach allows the designer to analyze each role individually); (2) roles focus on the notion of responsibilities, as opposed to classes that focus on capabilities (i.e. to analyze how objects collaborate, it is important to model the messages that objects must send in some circumstances, not just the messages that they can accept). A good overview of the importance of role models can be found in [8].

Our interest lies in 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. During our practice (consulting and development of case studies), we identified several modeling questions related to the utilization of use case diagrams that document system specification models.

The modeling questions we identified are related to the decomposition of a system into its subsystems, to the impossibility of specifying some important system requirements and to the reuse of use cases.

Some of these problems have already being identified by Ian Graham [3]. Desmond D’Souza and Allan Wills [1] provide a partial answer with their Catalysis method. In their method, they design a system by systematically going through all levels of abstraction (from business entities to “pluggable-parts” such as classes or components). This is done by action and operation refinements. The use case is used at each level of abstraction to specify what needs to be developed.

Our paper attempts 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, Sec-

tion 5: case study revisited using the extended use case modeling technique, Section 6: future work.

2 Modeling Questions

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

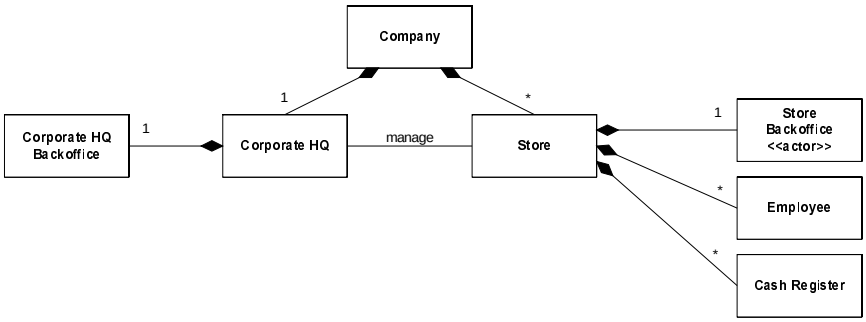

Fig. 1: Class diagram describing Company structure

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” (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 “download Price” (new price list need is downloaded from the Corporate HQ to all Cash Registers with the collaboration of the Store Backoffice).

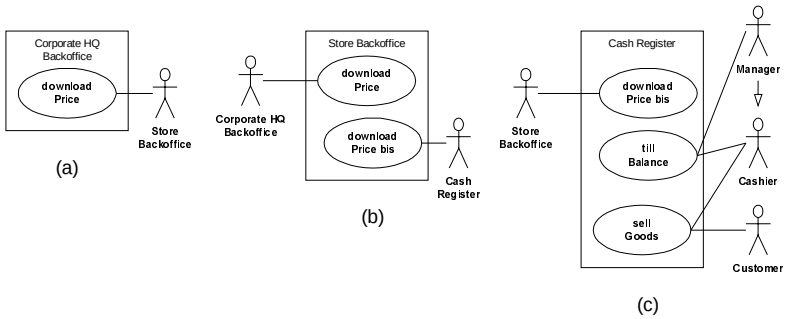

Fig. 2: (a) Corporate HQ Backoffice use case diagram, (b) Store Backoffice use case diagram, (c) Cash Register use case diagram

This example raises the following interesting points:

1. As the “download 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 systems to better understand the role of each system type relative to the other system types. We could use an interaction diagram. We reject this solution because it would force us to decompose the interactions between the systems down to the level of message 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: “download Price” and “download Price bis”. These two use case specifications are identical, except for the fact that, 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 (“download Price” and “download Price bis”). Of course, we want to have just one use case specification “download 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 “download Price“ use case involves just one recipient (unicast) or many of them (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 downloaded, “download Price” should occur first, followed by “download Price bis”. So use cases may have constraints between each other on when they may occur. Constraints may include for example sequentially, non-determinism, concurrency or real-time constraints [4]. 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 or as a system in the use case diagram (Fig. 2c and 2b) and as a class, possibly stereotyped with <<actor>>, in the class diagram (Fig. 1). The same concept is shown with a different symbol, 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 five identified questions and propose possible solutions.

To be precise, this paper will use the terms *type*, *class*, *specification*, *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 exam-

ple: 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. The terminology is consistent[1] with the RM-ODP definitions [4].

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 be able to model a business process that involves multiple system types and actor types, we need to be able (1) to indicate which system realizes which use case, and (2) to model the use cases involving only actors and no system. Currently UML use case diagrams, by either not representing the system or by representing only one system (drawn as a box around the use cases), this forces the designer to have only one system of interest in a use case diagram. In addition, the system-centric definition of use cases forces the modeler to represent only the use cases realized by one of the systems of interest. This leaves out the use cases not involving a system.

A possible answer can be found in Catalysis [1], a method that defines use cases as not system-centric. Their definition 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*". These smaller grained actions are themselves either joint actions or localized actions. A localized action is defined as an action involving only two participants (a sender and a receiver).

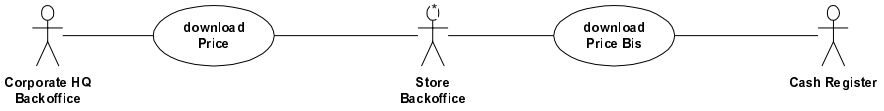

Fig. 3: Use case diagram representing systems with actors

By not referring to the system in their definition of use case, Catalysis allows specifying the collaboration of a group of entities (actors, sub-system or any other entities). This enables the representation of all the use cases of interest, regardless of the fact that the use case is realized or not by one of the systems of interest. Catalysis represents the use case participants with diagram elements corresponding to the actual entity (e.g. actor, subsystem, class, etc.).

By dropping the reference to the system, the use case definition provided by Catalysis solves our problem. A similar change of the UML definition of use case [class]

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

would read: "*the specification of a sequence of actions, including variants, that a group of entities performs when attempting to achieving some purpose*."

3.2 Reuse of Use Case Specifications

We are now addressing Question 2: "how can a system play different roles in different occurrences of a same use case specification?"

The Catalysis 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 "download Price bis" use case in Fig. 3). It is possible to use a collaboration concept to indicate that two use cases with different names are of the same type. However, this does not solve the problem of having two independent use case specifications and thus requires an additional diagram element.

The reuse problem comes from the fact that the use case specifications depend on their actors. We need to separate the use cases from the actors by using the concept of role. This is consistent with the UML definition of actors as a set of roles.

3.2.1 Introducing Roles

The UML definition of role is "*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).*" The second part of the definition is not important for our discussion. It is related to the future work and we ignore it in this paper. As for the first part, we question the fact that the behavior has to be named since roles in use case models are anonymous. An analysis of the definition of role can be found in [2].

So, in the context of use case modeling, we rephrase the role definition as: "a role is the specific behavior of an actor participating in a use case occurrence". This means that, with respect to a use case, an actor may be identified by a role it plays rather than by its name. Thus, roles provide the mechanism needed for making use case specifications independent of actors. This allows reusing use case specifications between different groups of actors and to have one actor instance playing different roles in different occurrences of the same use case specification.

Of course, we must have a mechanism to link roles to actors. In use case diagrams, our suggestion is to explicitly reference the actor's role on the association between the actor and the use case (see Fig. 4). We have defined two independent use case diagrams, as it is not clear if a same diagram can represent two use cases having a same name. This question is discussed in the next section.

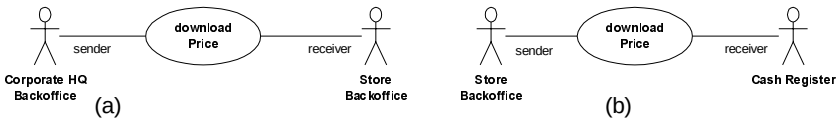

Fig. 4: Example of a same use case specification in two use case diagrams

3.2.2 Reusing Use Cases

UML specification does not seem to have rules imposing that all entities represented in one diagram should have different names. Thus it should be possible to represent the use case diagram shown in Fig. 5.

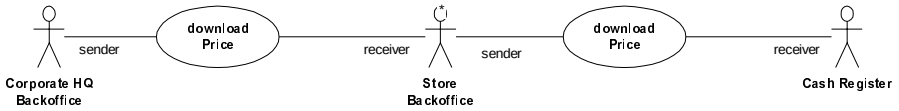

Fig. 5: Example of use case specification reuse in one use case diagram

However, even if UML allows having two “download Price” use cases in a same diagram, we still need to understand what these two identical elements actually represent. They model different classes of use case occurrences, but occurrences from a same use case specification. This is illustrated in Fig. 6. For example, the diagram element on the bottom left represents the classes of “download Price” occurrences between Corporate HQ Backoffice playing the role of sender and Store Backoffice playing the role of receiver. The element at the top represents the classes of all the “download Price” occurrences. But of course, we need different names for different classes. This is possible using a naming strategy analog to Smalltalk: we use the use case specification name to which we add contextual constraints (i.e. the role names and which actors play the roles). The class at the top is the class of all occurrences of the use case “download Price_sender< >_receiver< >”. The classes of the bottom left is a subset of the class at the top; it represents all the occurrences where the sender is of the type Corporate HQ Backoffice and the receiver is of the type Store Backoffice.

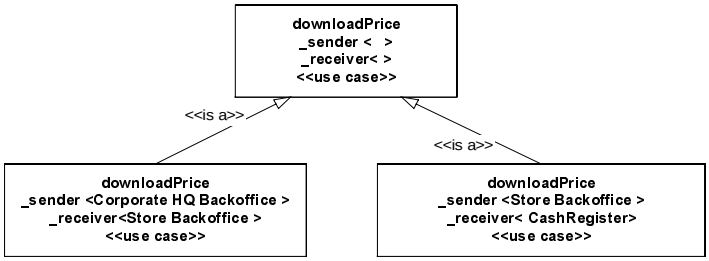

Fig. 6: Naming scheme for classes of use case instances of a same specification

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

3.3 Constraints about Instances

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

It is not clear if 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 models having multiplicities. 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 element.

The cause of this problem relates to the relationship between types and instances. These terms are frequently mixed as illustrated in the following two examples "*roles (in collaborations) are somewhat between types and instances*" [9] and "*if there can be more than one instance corresponding to a given ClassifierRole, one of these instances is selected to represent them all*" [10]. 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) [7]. 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 policeman in uniform is considered as defining a type (i.e. the policeman) but is an instance at the same time (i.e. the man currently in the middle of the crossing). This concept of prototype explains why, sometimes, concepts are difficult to categorize as type or instance. Unfortunately, when developing models as the ones defined in UML, the prototype concept is not workable. 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 (defined as an occurrence of something at a specific location in time and space)
2. Types are tools used to categorize instances into classes
3. Instances are useful for formalizing interactions between concepts
4. Classes are useful for thinking and working with instances

So we propose to define all UML concepts as type and instance (this is already the case for most of them). In the context of use cases, we propose to add the concepts of *role instance* (role type is already defined), as well as, *actor instance* (actor types are already defined). Note that, as the use case instance already exists, our propositions would remove some exceptions in the meta-model. Acknowledging the existence of actor instances in UML enriches the use case diagram by enabling the expression of numerical constraints on the number of actors participating in a use case occurrence. This is illustrated in Fig. 7.

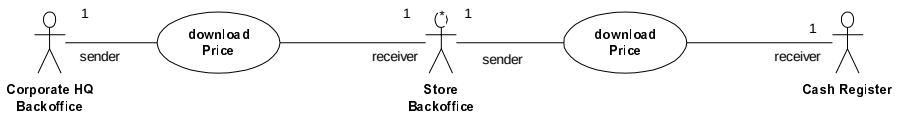

Fig. 7: Example of multiplicity in use case diagram

Again, the notation is analog to the class diagram notation. The multiplicity is written only on the actor side of the association, as the multiplicity expresses constraints on the number of instances involved in one use case occurrence.

The use of multiplicities in use case diagrams can clarify significantly the use case specifications when the use cases are used for specifying distributed systems. A use case diagram without multiplicity describes only system types. To understand the distribution, it is important to provide information on the system instances (of a same type), hence to have multiplicities. This allows documenting issues such as unicast or multicast.

3.4 Constraints on Use Case Occurrences

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

Having role instance and /actor instance concepts enables having use case instance diagrams. These diagrams model occurrences of use cases and instances of actors. By numbering the occurrences of use cases, it is then possible to define 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. Fig. 8 shows an example. Note the anonymous message sent by the actor to it. This allows identifying clearly who is responsible for the multicast.

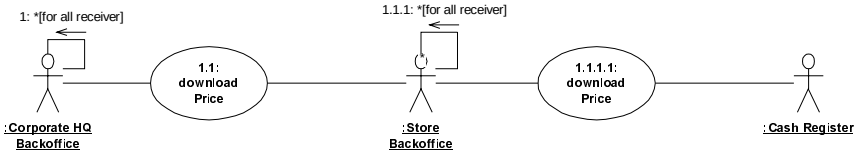

Fig. 8: Example of use case instance diagram

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*". We have the following problem with this definition: all entities realize a set of roles. So what is so unique to the actor? To understand the specificity of the actors, we need to consider how they are used:

1. An actor links use cases together by realizing a set of roles. For example, in Fig. 7, the Store Backoffice actor receives the prices by participating in a class of "download Price" use cases and then sends these prices by participating in a second class of "download Price" use cases.

2. 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. 7, the Store Backoffice actor represents the Store Backoffice shown in the class diagram in Fig. 1.
3. An actor may have a generalization relationship with another actor. For example, in Fig. 2c, Manager is a generalization of Cashier. That is, all managers are also cashiers.
4. An actor represents entities that will not be further specified. For example, in Fig. 2c, Customer actor represents a person coming in the Store to purchase goods. The designer will not need to specify it further as there is no need to know more about the entity realizing the behavior of the Customer.

The first use does not require a specific actor concept. Any entities realize a set of roles, so any entities could be used in a use case diagram to link two use cases together. Only the definition of the use case diagram forces the systematic use of actors.

The second 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), the need for actors in use case diagrams would largely disappear. The resulting diagram is apparently more complex (more type of entities are represented) but is actually simpler to use (as it removes unnecessary indirection to the other diagrams). The Fig. 9 illustrates the use of different entities in the use case diagram. Catalysis uses a similar representation.

Fig. 9: Example of use case diagram representing actual entities as opposed to actors

Symmetrically, it should be allowed to represent actor types in all diagrams representing class (e.g. class diagram) and actor instances in all diagrams representing objects (e.g. collaboration diagram). This would eliminate the <<actor>> stereotype (as it is the same diagram element which is used in all diagrams). It would also be consistent with the use of actors in interaction diagrams. The actor is usually used to represent the entity initiating the interaction.

The third use of actors is to show generalization relationships between participants to use cases. In our example, the generalization relationship between Manager and Cashier is intended to mean that a manager may perform the roles of a cashier. It is not intended to mean that the role of the Cashier in “sell Goods” is a Manager’s role (just that it is a role that a manager can fulfill). This shows that we do not think as actors as sets of roles, otherwise all roles inherited from Cashier would be Manager’s roles.

The fourth use illustrates the specificity of the actor concept over the other entity concepts. An actor is used when the designer is interested in modeling an entity having a specific behavior. This can be useful for (1) modeling entities participating in the

business process but which are not one among the systems of interest, and (2) modeling patterns of collaboration in which the designer does not want to specify what entity will realize the role. When used in patterns of collaboration, the actor might not even have a name (as its role is already identified).

Our discussion implies that Actor [class] should not be defined as a set of roles but rather as "*the partial specification of an entity; this partial specification is obtained by composing the roles that the entity performs in use cases or in collaborations*". As discussed in [2], a role is itself an abstraction (or partial specification) of an entity, but with respect to a single use case or collaboration. If absolutely all the roles that an entity performs are specified for an actor, the actor is equivalent to the entity itself. This confirms that Actor concepts can be considered as first class modeling concepts and can be found in any diagrams.

4 Modifications to UML

In this section we discuss the impact of our proposal on UML. The main changes are related to the definitions:

Use case [class] - the specification of a sequence of actions, including variants, that a group of entities performs when attempting to achieve some purpose.
Use case [instance] – an occurrence of a sequence of actions as specified in a use case [class].
Actor [class] – the partial specification of an entity; this partial specification is the composition of the roles that the entity performs in use cases or in collaborations.
Actor [instance]- an instance of an actor [class].

Simply by changing these definitions, we address most of the modeling questions that we raised. This implies that current UML case tools can, in principle, can be used unchanged for applying our extended modeling technique. The only issue is whether the use case tool will complain when a modeler uses a same use case specification twice in a use case diagram (reuse).

To be able 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[2] (see Fig. 10).

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

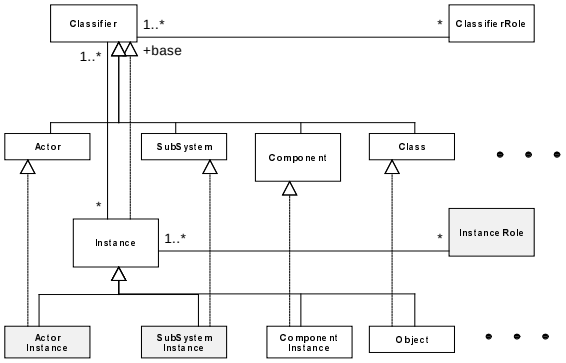

Fig. 10: Elements of meta-model related to the classifier – instance relationship

The proposed modifications make the model more consistent as they remove exceptions (Classifier concepts without corresponding Instance concepts).

5 Application of our Suggestions

We were motivated by an experience in a real IT project. In Section 2, we presented "classical" models defined in that project (Fig. 1 and Fig. 2). We now present analog models that reflect the use of our new definitions of use case and actors (see Fig. 11 and Fig. 12).

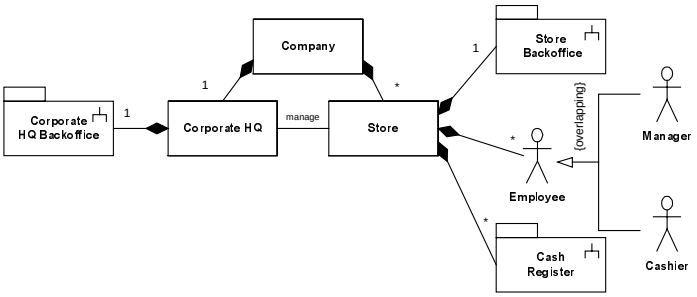

Fig 11: Class diagram describing Company organization

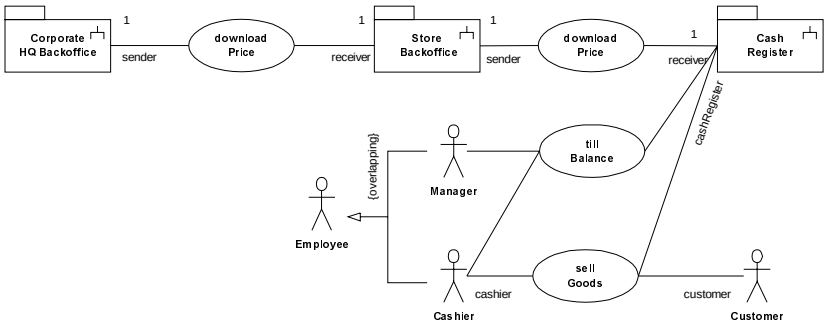

Fig 12: Use case diagram describing the business processes to be implemented

Our new use case modeling technique is essentially compatible with the current technique. For examples, specifying roles explicitly is optional (in Fig. 12 the roles in "till Balance" are not specified).

Showing the systems explicitly as actors tends to lead to situations where an actor is associated to many use cases. This makes drawing use case diagrams quite cumbersome. To address this problem, we introduce the notation rule that an actor can be represented by a rectangle around a set of use cases. Again this notation rule is quite compatible with the current use case modeling technique. The only difference is that the box represents any actor and the systems have to be represented (either as a box or an actor).

Sometimes, roles are difficult to name. As the roles are bound to the use case, it is possible to use the same identifier to denote an actor and its roles in the use cases. The identifier might start with a capital letter when denoting an actor and a lower case when denoting the role. In our example in Fig. 12, Cashier as the "cashier" role in the "sell Goods" use case. This convention is consistent with the one used in class diagrams to denote AssociationEndRole.

In our new model in Fig. 12, we no longer use an inheritance relationship between Manager and Cashier but an overlapping generalization relationship between Employee and Manager and Cashier. This reflects the fact that not all managers will necessarily be cashiers (for big stores, we might even have a policy that managers will not be cashiers). We are taking advantage here of our new definition of actor. Note the consistency between the use case diagram in Fig. 12 with the class diagram in Fig. 11.

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. A consequence of these changes is that all entities may be represented in all UML diagrams. Moreover, the notational techniques that we propose for use case diagrams were borrowed from those of class diagrams and collaboration diagrams. We believe that this is more than a mere coincidence: these diagrams are essentially different notational techniques for a same model,

and 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, (3) simplification of case tools.

Conclusion

Use cases are the modeling technique of UML for formalizing the functional requirements placed on systems or on businesses. 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 model). 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 can-not 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 two principles: treating the system as an ordinary actor, and making the roles of actors explicit in models (that is, we propose to specify use cases with role names rather than with actor names). These two principles imply very limited changes to UML: the definitions of actor and use case must be revised, and the reuse of use cases must be allowed between different groups of actors.

A complementary idea is to enable modeling at the level of use case occurrences and actor instances (the diagrammatic techniques being borrowed from those of interaction diagrams). We think that modeling at this level is invaluable for relating use cases together, or for expressing execution constraints between them. The changes required 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 go in the direction of increasing 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 and Frederic Bouchet (Compaq Professional Services, Geneva, Switzerland) helped to identify the problems of modeling the reengineering of a business process at the abstraction level of IT systems.

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, "A Foundation for the Concept of Role in Object Modelling," EPFL Technical Report (in preparation)
3. Graham Ian, Requirements Engineering and Rapid Development: an object-oriented approach, Addison-Wesley, 1998, ISBN 0-201-36047-0.
4. 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).
5. Jacobson Ivar, Christerson Magnus, Jonsson Patrick, Övergaard Gunnar*: Object-Oriented Software Engineering*, Addison-Wesley, 1992, ISBN 0-201-54435-0
6. Jacobson Ivar, Ericsson Maria, Jacobson Agneta*: The Object Advantage – Business Process Reengineering with Object Technology*, Addison-Addison Wesley, ISBN 0-201-42289-1
7. Lakoff George: Women, *Fire and Dangerous Things – What Categories Reveal about the Mind*, Chicago Press, 1987, ISBN 0-226-46804-6
8. Li Qing, Wong Raymond: "Multifaceted object modeling with roles: A comprehensive approach," in Information Sciences 117 (1999) 243-266, Springer Verlag
9. OMG, *Unified Modeling Language Specification*, Version 1.3, June 1999, www.omg.org
10. Reenskaug Trygve, "UML Collaboration and OOram semantics – New version of green paper," 2nd ed, Nov. 8, 1999 (http://www.ifi.uio.no/~trygver/documents)
11. Reenskaug Trygve, Wold Per, Lehne Odd Arild, *Working With Objects: The OOram Software Engineering Method*, Manning Publications, 1996, ISBN 0-13-452930-8