Generalization Relation in UML Model Elements

Claudia Pons

LIFIA – Laboratorio de Investigación y Formación en Informática Avanzada
Universidad Nacional de La Plata, CP 1900 Buenos Aires, Argentina
email:cpons@info.unlp.edu.ar

Abstract

In this paper we analyze the concept of Generalization in the UML metamodel. We revise the kinds of generalization/specialization relations between UML model elements that can be expressed. We arrive to the conclusion that the Generalization relationship provided by the UML actually embraces two different generalization/specialization relations: an *incremental* relation and an *overriding* relation. Nature of each relation is very different, and it must be clearly distinguished in the UML in order to avoid wrong interpretation of UML models.

Keywords: Object Oriented Analysis and Design, Modeling Languages, Unified Modeling Language, Generalization relation, inheritance.

1. Specialization Mechanisms in the Object-Oriented Paradigm

In the Object Oriented paradigm [Booch 94], a subclass describes the structure and behavior of a set of objects. However, it does so incrementally, by describing extensions and changes to its direct superclasses. A subclass is completely compatible with its superclass if the subclass has the same argument domain as the superclass and, for all operations of the superclass, corresponding arguments yield corresponding results. Complete compatibility permits instances of the subclass to be freely manipulated as instances of the superclass with no fear of inadmissible behavior. This concept is called principle of substitutability.

In actual programming languages there exist two different mechanism to obtain a subclass from a given class by modifying its signature and/or behavior: *Incremental specialization* and *overriding specialization*. In general, these mechanisms do not guarantee substitutability, because evolutionary processes of incremental change in the real world rarely conform to the stringent constraints of complete compatibility.

Incremental (or horizontal) specializations are subclasses that specialize the parent class having a richer set of messages than the parent class. It means that messages defined for both the subclass and the superclass have identical signature and semantics. However, the subclass may have additional messages. Figure 1 shows an example of horizontal extension of Class Point as follows:

Point.attributes ={x,y}

ColorPoint.attributes = {color}

ColorPoint.allAttributes = {x, y, color}

Overriding (or vertical) specializations occur when an overriding method is allowed to adopt different argument, result types and behavior (when behavior is too difficult to specify it can be approximated by a signature. In particular, type-checking algorithms usually define types in terms of signatures rather than behavior). For example method *#equal* in figure 1. There are three different forms of overriding: invariant (the domain remains the same), co-variant (the domain is restricted), contra-variant (the domain is loosened). Only modifications which define restrictions of the result of methods and/or define generalizations of the arguments of methods can satisfy the principle of substitutability. Type safety requires that the type of the result of the overriding method be a subtype of the type of the result of the overridden one, and that the type of the parameters of the overridden method are subtypes of those of the corresponding parameters in the overriding one. In type theory, this rule is said to be covariant in the result type (since it preserves the sense of the subclass relation) and contravariant on the parameters' types (since it inverts the sense of the subclass relation).

But, contravariant specialization of methods restricts the flexibility of a language considerably because in object oriented programming the most frequent vertical modification of signatures consists in defining restrictions of both the arguments and the range of methods. For this reason, most object-

oriented languages adopt a covariant specialization rule, which requires that the type of the parameters in the overriding method are subtypes of those of the overridden one. This approach is more flexible but does not ensure type safety (i.e. the relationship between subclasses defined in this way does not conform to the principle of substitutability). In the last years several solutions have been suggested to obtain this flexibility without losing type safety [Castagna95, Bruce95, Abadi96, Boyland96].

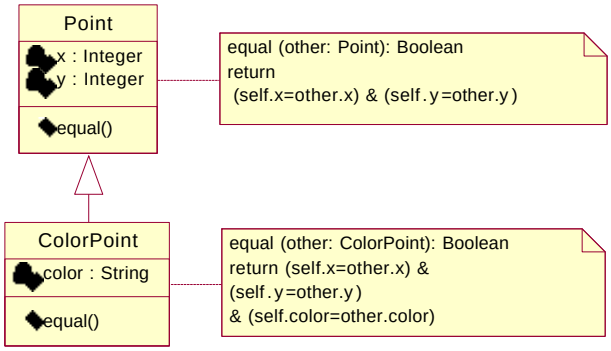


Figure 1: incremental (horizontal) and overriding (vertical) specialization

The problem with the covariant specialization of a method m: A → B with m': A'→B' and A'⊆ A is that m' cannot handle some arguments that m can (i.e. the elements in the set (A - A') are not handled by m'). Considering the class hierarchy depicted in figure 1, and the following assignments to variables p2 and p3: p2:= new(Point) . p3:=new(ColorPoint). The expression p3.equal(p2) can be obtain by subsumption (using p3 in a context where a Point was expected) and the object p2 will be asked for its color rising a wrong behavior.

2. Generalization/specialization mechanisms in the UML

At the present the Unified Modeling Language [UML 01] is considered the standard modeling language for object oriented software development process. The specification of UML constructs and their relationships is semi-formal, i.e. certain parts of it are specified with well-defined languages while other parts are described informally in natural language. There is an important number of theoretical works giving a precise description of core concepts of UML and providing rules for analyzing their properties. See for instance the works of [Evans et al.,1998;1999], [Kim and Carrington, 1999], [Breu et al., 1997], [Övergaard , 1998, 1999, 2000], [Pons and Baum, 2000], [Giandini et al., 2000], [Pons et al., 2000]. But, several UML concepts still need deeper analysis and formalization. In this article we focus on UML generalization/specialization hierarchies.

The UML provides a graphical construction, called Generalization and depicted by an arrow (see figure 1), to specify generalization/specialization hierarchies. According to the UML specification [UML 01], a Generalization denotes a taxonomic relationship between a more general element and a more specific element. The more specific element is fully consistent with the more general element (it has all of its properties, members, and relationships) and may contain additional information. As a consequence of this definition we can clearly see that UML considers Generalization as an incremental specialization mechanism.

The UML semantics document explains the meaning of the Generalization construct in terms of segment descriptors. A full descriptor is the full description needed to describe an instance. It contains a description of all the attributes, associations, and operations that the instance contains. In OO languages, the description of an instance is built out of incremental segments that are combined using inheritance to produce a full descriptor for an instance. The mechanism of inheritance defines how full descriptors are produced from a set of segments connected by generalization.

In the UML metamodel the prefix ***all*** is used to denote inherited features. For example, the additional operation *allFeatures* in the metaclass Classifier, results in a set containing all Features of the Classifier itself and all its inherited Features, as follows:

allFeatures : Classifier -> Set(Feature)

allFeatures = self.features ->union (self.parents.allFeatures)

In the UML, a GeneralizableElement is a model element that may participate in a Generalization relationship. As expected, UML defines that a Class is a GeneralizableElement, but also UML considers that other model element, such as Association, Stereotypes, Signals and Use Cases, may be treated as GeneralizableElements.

The concept of generalization/specialization hierarchy is well understood when it is applied on Classes and in general it is compatible with the concept of incremental inheritance hierarchy. In other words, any element in a generalization/specialization hierarchy is considered as an increment of its parents. But, if we try to keep this compatibility with incremental inheritance, when the concept of generalization/specialization hierarchy is extended to other model elements several contradictions and ambiguities arise.

The objective of this paper is to analyze UML generalization/specialization hierarchies in order to discern which kind of UML model elements can (properly) participate in a generalization/specialization hierarchy and which is the role each model element plays in the hierarchy.

3. Generalization Hierarchy of Associations

In the UML metamodel the metaclass Association is a child of the metaclass GeneralizableElement. According to the definition of inheritance, it is expected that a child association inherits some properties from its parents, but this is not the case: the only additional operation with prefix ***all*** is allConnections that results in the set of all AssociationEnds of the Association itself. It is defined as follows (see that connections belonging to parents are not considered at all):

allConnections: Association -> Set(AssociationEnd)

allConnections = self.connection

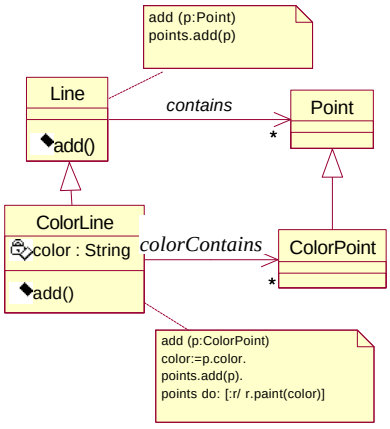


Figure 2: specialization of Association

Let's see an example. In figure 2 we show a UML model specifying four classes: a Line that consists of a sequence of Points and a ColorLine (specializing Line) that consists of a sequence of ColorPoints (specializing Point). The Association named "contains" linking a Line with one or more Points is then specialized by the Association named "colorContains" in order to specify that a ColorLine can only be connected to ColorPoints.

Analyzing this example, we see that the UML definition of allConnections is reasonable because child Association does not inherit parent's properties (such as parent's connections); child association redefines parent's connections. That is to say:

contains.connections={e1,e2}

where e1.type=Line and e2.type=Point

colorContains.connections={e3,e4} where e3.type=Colorline and e4.type=ColorPoint.

It makes no sense that Association *colorContains* inherits parent's connection between Point and Line, for example if *colorContains* inherited parent's connection, it would become a quaternary association.

Semantically the generalization/specialization relationship between associations denotes an inclusion relation between the corresponding intentional sets, where the intentional set of an association consists of all potential instances of that association. That is to say, let A and B be Associations, if A is a specialization of B, then A.instances$\subseteq$ B.instances.

In the simple case in which the association is a binary one, its potential instances are pairs of classifier instances, in the following way:

instances: Association -> Set (Instance x Instance)

A.instances = { (x,y) | x $\in$ A.source.instances and y $\in$ A.target.instances }

Where *source* (respectively *target*) is a function that returns the Classifier that is source (respectively target) of the binary Association. And *instances* is a function that applied on a Classifier returns the set of all potential instances of that Classifier. For example:

contains.instances ={ (x,y) | x ∈ Line.instances and y ∈ Point.instances }

colorContains.instances ={ (x,y) | x ∈ ColorLine.instances and y ∈ ColorPoint.instances }

On the other hand specialization of Association always occurs together with specialization of AssociationEnds´s types. That is to say, the inclusion relation A.instances⊆B.instances only can be proved to hold if both A.source.instances⊆ B.source.instances and A.target.instances⊆ B.target.instances (this property holds if A.source is a subclass of B.source and A.target is a subclass of B.target).

For example, it is straightforward to prove that the set colorContains.instances is contained into the set contains.instances as follows:

colorContains.instances=

{ (x,y) | x ∈ ColorLine.instances and y ∈ ColorPoint.instances }

⊆ { (x,y) | x ∈ Line.instances and y ∈ ColorPoint.instances}
because ColorLine.instances ⊆,Line.instances

⊆ { (x,y) | x ∈ Line.instances and y ∈ Point.instances}
because ColorPoint.instances ⊆,Point.instances

= contains.instances

In conclusion, Association specialization is quite different from Class specialization, Indeed, it looks like method specialization. That is to say, generalization/specialization relationship between Associations is an ***overriding*** relationship, not an ***incremental*** relationship.

Coincidentally, when we specialize Associations we face the same problems that arise when we specialize Methods. That is to say, covariant specialization (such as the one in figure 2)is the most useful way of Association specialization but is not safe. Instances of ColorLine cannot be used in any context in which an instance of Line was expected., as we can see in the following example, after the following assignments to variables c1 and p: cl:= new(ColorLine). p:=new(Point). The expression cl.add(p) can be obtain by subsumption (cl is used in a context were instances of Line were expected) and the object p will be asked for its color rising a wrong behavior..

3.1 Well formedness rules of Association Hierarchies

The UML specification document [UML 2001] includes a few constraints regulating the form of Association Hierarchies. But it is necessary to define more constraints in order to avoid ambiguities.

Since child Association specializes their parents, it is natural to say that each AssociationEnd of a child Association specializes the corresponding AssociationEnd (the AssociationEnd in the same position) in its parents. To reflect this fact, it is necessary to add new well formedness rules, expressed in OCL [UML 2001] in the UML metamodel. The following rule on Association is an example of this kind of rules:

1. The property *aggregation* of each AssociationEnd in a child Association should maintain or restrict the aggregation of AssociationEnds in the same position in its parents. Possible values that can be assigned to property aggregation are: #none, #aggregate and #composite, where #composite restricts to #aggregate, and #aggregate restricts to #none.

```
self.parent -> forAll
( p | Sequence {1..p.connection -> size}  ->
   (      (p.connection -> at(i).aggregation = #composite implies
                 self.connection->at(i).aggregation = #composite)
          or (p.connection -> at(i).aggregation = #aggregate implies
                 (
                   (self.connection->at(i).aggregation = #aggregate)
                  or (self.connection->at(i).aggregation = #composite)
                 ))  )
```

Without these additional rules it would be possible to create ill-formed UML models that violate the basic principles of the generalization/specialization concept.

4. Generalization Hierarchy of AssociationClass

AssociationClass is a very special model element, because it is at the same time both a Class and an Association. Then, in order to understand specialization of AssociationClass we need to conciliate the dichotomy between both different kinds of specialization. Lets see what the UML does.

The operation allConnections is redefined in the metaclass AssociationClass resulting in the set of all connections of the AssociationClass, including all connections defined by its parents:

```
allConnections: AssociationClass -> Set(AssociationEnd)
allConnections = self.connection -> union
                ((self.parent->select (a | a.OclKindOf(Association))) ->collect (a | a.allConnections))
```

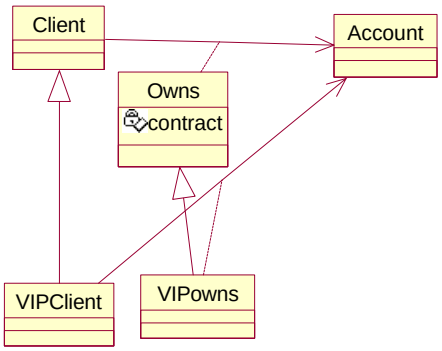


Figure 3: specialization of AssociationClass.

This definition is inconsistent with the definition of allConnections in Association, because an Association does not inherit its parent's connections.

On the other hand it seems no possible to specialize the AssociationEnds of an AssociationClass (as we can do with Associations). In figure 3 we can see an example where the AssociationClass *Owns* is specialized by the AssociationClass *VIPowns*. But also we could be interested in restrict the kind of Client that can subscribe this VIP association. However, according to the definition of allConnections above, we cannot prevent *VIPowns* from inherit the connection with Client, as follows:

owns.allConnections={e1,e2}, where e1.type=Client and e2.type=Account

VIPowns.allConnections={e1,e2,e3, e4} where e3.type=VIPClient and e4.type=Account

We propose to keep the possibility of defining inheritance relationship between AssociationsClass, but modifying the definition of the additional operation allConnections in the following way:

allConnections: AssociationClass -> Set(AssociationEnd)

allConnections = self.connection

This new definition is consistent with the definition of specialization between Associations and also it avoids the problem of inheriting unwanted connections. For example, with the new definition *Vipowns*.allConnections results in the set {e3,e4}, that is the intended meaning of the diagram.

5. Specializing actors

An Actor is a UML element that defines a coherent set of roles that users of an entity can play when interacting with the entity. In the metamodel Actor is a GeneralizableElement. Therefore an Actor may have generalization relationships to other Actor. This means that the child Actor will be able to play the same roles as the parent Actor (i.e. communicate with the same set of UseCases as the parent Actor). On the other hand, a child UseCase inherits all Associations to Actors of the parent, and may add new ones.

When combining the specialization of UseCases with the specialization of actors, confusing situations can arise. In the example in figure 4 (taken from [Cockburn 01]) the model tries to express the idea that a Sales Clerk can close any deal, but that it takes a special kind of sales clerk, a Senior Agent, to close a

deal about a certain limit. Cockburn says that actually it does not express this properly, since the UML rule is “A specialized use case can be substituted wherever a general use case is mentioned”. Therefore, the drawing says that an ordinary sales clerk can close a big deal too.

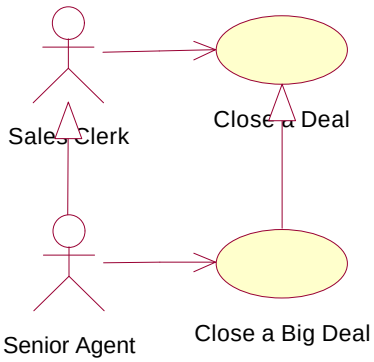


Figure 4: specializing actors

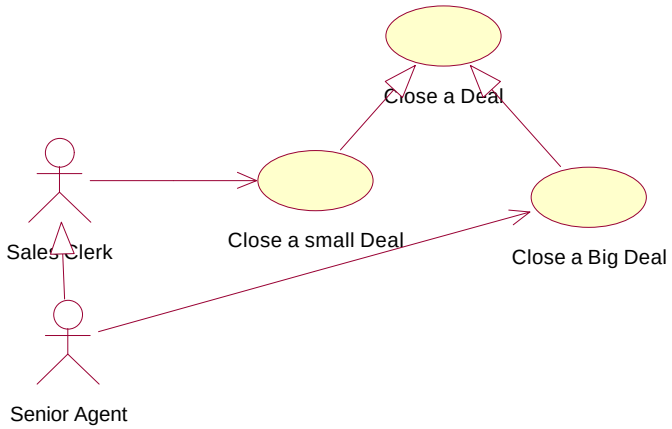
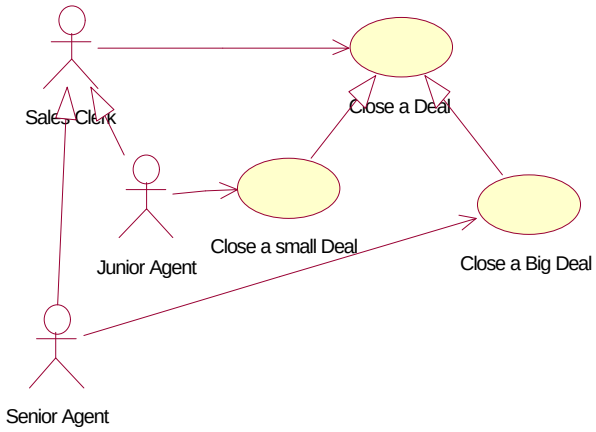


Figure 5: solution #1

Actually the problem exists, but not as a consequence of the specialization of the UseCase. Lets see why. Lets consider the set of links that may potentially instanciate the Association between Actor SalesClerk and UseCase CloseADeal:

SalesClerk-CloseADeal.instances={ (x,y) | x∈SalesClerk.instances and y∈CloseADeal.instances }

But JuniorAgent.instances is included into SalesClerk.instances, because JuniorAgent is a specialization of SalesClerk. Additionally, CloseABigDeal.instances is included into CloseADeal.instances because CloseABigDeal is a specialization of CloseADeal,. Therefore any pair (a,b) with a∈JuniorAgent.instances and b∈CloseABigDeal.instances may instanciate the Association leading to a wrong behavior. Then Cockburn in [Cockburn 01] proposes the drawing in figure 5 to solve the problem. Technically this a solution, but the actor in the parent use case is missing., therefore it is not possible to allocate behavior to the parent UseCase because Actor drives the behavior of the UseCase.



Use Case Close a Deal

1. System checks the authorization of the SalesClerk to Close de deal.

2. If authorization is OK, close the deal.

Use Case Close a Big Deal

1. Action checks the authorization is refined as: if SalesClerk is a Senior then authorization is OK, otherwise authorization is not OK

2. Action close the deal is refined as: system ask for references about the client. If references are OK, then sign the contract and record the deal into the files.

Use Case Close a Small Deal

1. Action checks the authorization is refined as: for any SalesClerk authorization is OK.

2. Action close the deal is refined as: sign the contract and record the deal into the files.

Figure 6: solution #2

Having in mind our idea respect to generalization hierarchy of Associations, we propose an other solution in figure 6. The later model expresses a covariant specialization of the Association between Actor and UseCase. The situation is similar to covariant method overriding we discussed in section 2.

That is to say, the child UseCase does not restrict the type of Actor involved in the UseCase, instead it specializes the behavior over a subset of the domain of SalesClerks and keeps the old behavior over the rest of the domain. Note that parent UseCase can deal with both kind of SalesClerk, but the behavior is specified in an abstract way, then each child specializes that abstract behavior for a specific kind of Actor.

6. Conclusions

The objective of this paper was to analyze which kind of model elements can (properly) participate in a generalization/specialization hierarchy. We got to the conclusion that the Generalization relationship provided by the UML actually embraces two different generalization/specialization relationships: an ***incremental*** relationship (i.e inheritance) and an ***overriding*** relationship (i.e redefinition). Nature of each relationship is very different, and it must be clearly distinguished in the UML in order to avoid wrong interpretation of UML models.

In the UML semantics document Generalization relationship is described as an incremental specialization mechanism. This description is appropriate for the usage of Generalization in connection with Classes, whereas the usage of Generalization whit other model elements, such as Association, Stereotypes, Signals and Use Cases, is better described as an overriding specialization mechanism or as a combination of both mechanisms.

The analysis reported in this article contributes to the improvement of UML syntax and semantics. Formalization of the UML is an important task because the lack of accuracy in its definition causes wrong model interpretations and discussion regarding the model meaning. The interpretation done by the people who reads the model may not coincide with the interpretation of the model creator. These misunderstandings lead to the highly expensive problem of construction of systems that do not meet user expectations.

References

Abadi,M and Cardelli,L. A Theory of Objects, Springer Verlag. 1996.

Booch, G.,Object Oriented Analysis and Design with Applications. Addison-Wesley , 1994.

Bruce,K van Gent, R and.Schuett, A., PolyTOIL: A Type-Safe Polymorphic Object-Oriented language, in proceedings 9th European Conf. on O-O Programming (ECOOP'95), Springer, LNCS 952, 1995.

Boyland,J and .Castagna, G.,Type-safe compilation of covariant specialization: a practical case, proceedings 10th European Conf. on O-O Programming (ECOOP'96), Springer, LNCS 1098, 1996.

Cook, W, Hill, W. and Canning, P Inheritance is not subtyping, In Theoretical Aspects of OO Languages, MIT Press, 1994.

Castagna, G Covariance and Contravariance: conflict without a cause. ACM Transactions on Programming Languages and Systems, 17(3), 1995.

Cockburn, Alistair. Writing Effective Use Cases. Addison-Wesley. 2001

Breu,R., Hinkel,U., Hofmann,C., Klein,C., Paech,B., Rumpe,B. and Thurner,V., Towards a formalization of the unified modeling language. ECOOP'97 procs., Lecture Notes in Computer Science vol.1241, Springer, (1997).

Evans,A., France,R., Lano,K. and Rumpe, B., Developing the UML as a formal modeling notation, UML'98 Beyond the notation, Lecture Notes in Computer Science 1618, Springer-Verlag, (1998).

Evans,A., France,R., Lano,K. and Rumpe,B., Towards a core metamodelling semantics of UML, Behavioral specifications of businesses and systems, Kluwer Academic Publishers, (1999).

Giandini,R, Pons, C and Baum,G.. An algebra for Use Cases in the Unified Modeling Language. OOPSLA´00 Workshop on Behavioral Semantics, Minneapolis, USA, October 2000.

Kim, S. and Carrington,D., Formalizing the UML Class Diagrams using Object-Z, proceedings UML´99 Conference, Lecture Notes in Computer Science 1723, (1999).

Övergaard, G., and Palmkvist,K., A Formal Approach to Use Cases and Their Relationships. In P. Muller and J. Bézivin editors, Proceedings of the UML'98: Beyond the Notation, Lecture Notes in Computer Science 1618. Springer-Verlag, (1998).

Övergaard, G., A formal approach to collaborations in the UML, <<UML>>´99 - The Unified Modeling Language. Beyond the Standard, Lecture Notes in Computer Science 1723, (1999).

Övergaard,G.. Using the Boom Framework for formal specification of the UML. in Proc. ECOOP Workshop on Defining Precise Semantics for UML, France, June 2000.

Pons Claudia and Baum Gabriel. Formal foundations of object-oriented modeling notations 3rd International Conference on Formal Engineering Methods, ICFEM 2000, York, UK.IEEE Computer Society Press. September 2000.

Pons, Claudia, Giandini Roxana and Baum Gabriel. Specifying Relationships between models through the software development process, Tenth International Workshop on Software specification and Design (IWSSD), San Diego, California, IEEE Computer Society Press. November 2000.

UML, The Unified Modeling Language Specification – Version 1.4, UML Specification, revised by the OMG, http://www.omg.org, September 2001

Wegner, P.and.Zdonik, S, Inheritance as an Incremental Modification Mechanism or What like is an isn't like. in proceedings 3rd European Conference on O-O Programming (ECOOP'88), Springer, 1988.