Metamodeling generalization and other directed relationships in UML

Gonzalo Génova¹,², Juan Llorens², Anabel Fraga²

¹Facultad de Ingeniería, Universidad Andrés Bello
Sazié 2325, Santiago, Chile
gonzalo.genova@unab.cl
²Departamento de Informática, Universidad Carlos III de Madrid
Avda. Universidad 30, 28911 Leganés (Madrid), Spain
{ggenova, llorens, afraga}@inf.uc3m.es

Abstract. Context - Generalization is a fundamental relationship in object orientation and in the UML (Unified Modeling Language). The generalization relationship is represented in the UML metamodel as a “directed relationship”. Objective - Being a directed relationship corresponds to the nature of generalization in the semantic domain of object orientation: a relationship that is directed from the subclass to the superclass. However, we claim that the particular form this relationship adopts in the metamodel is erroneous, which entails a series of inconveniences for model manipulation tools that try to adhere to the UML specification. Moreover, we think that this error could be due to a misinterpretation of the relationships between metamodeling levels in the UML: represented reality (M0), model (M1) and metamodel (M2). This problem also affects other directed relationships: Dependency and its various subtypes, Include and Extend between use cases, and others. Method - We analyze the features of the generalization relationship in various domains and how it has been metamodeled in UML. We examine the problems, both theoretical and technological, posed by the UML metamodel of generalization. We then compare it with the metamodel of other directed relationships. Results - We arrive at the conclusion that the metamodel of all directed relationships could be improved. Namely, we claim that, at level M2, the metamodel should not contain any one-way meta-associations; all meta-associations should be two-way, both for practical and theoretical reasons. Conclusions - The rationale for our main claim can be summarized as follows: connected graphical symbols do know each other, and the goal of a metamodel is to specify the syntactic properties of a language, ergo meta-associations must be two-way. This, of course, does not preclude at all the use of one-way associations at the user model level (M1).

Keywords. Unified Modeling Language; Model Engineering; Metamodel; Generalization; Directed Relationship.

1 Introduction

As it is well known, the generalization, or inheritance, relationship is one of the basic concepts of object oriented programming. Generalization (or better its counterpart, specialization) allows defining subclasses of a given class, so that: (1) the superclass is defined in a totally independent way of the extensions that may happen
in the subclasses, such as adding attributes or operations, or even redefining operations; and (2) client classes of the superclass are defined also in a totally independent way of the potential subclasses. These two characteristics are one of the bases for the definition of polymorphic behaviors, that is, given the invocation of an operation from the client class, the possibility of reacting with different behaviors selected at run time. Of course, all of these different behaviors must satisfy the same operation contract, following the discipline of design by contract [15] and the substitution principle [13].

Consequently, generalization in object oriented programming is a kind of one-way relationship, where the origin of the relationship (the subclass) knows the target of the relationship (the superclass) but, conversely, the superclass does not know any of its subclasses; it does not even know whether they exist. In any object oriented programming language the relationship is expressed exclusively in the subclass, by means of some kind of keyword or symbol (such as ‘extends’ in Java, ‘:’ in C# or ‘<’ in Ruby), whereas the code of the superclass does not express the relationship at all. When the subclass is compiled, it is a requirement that the superclass exists and is known to the compiler, so that compatibility can be checked; obviously, there is no similar restriction when compiling the superclass (the same applies to interpreted languages like Ruby). In UML (Unified Modeling Language) [21], and other similar modeling languages, this asymmetric directionality of the relationship is graphically expressed by means of the familiar “generalization arrow”, where source and target classes are clearly indicated. Similarly, other directed relationships are also represented with different kinds of arrows: Dependency and its various subtypes, Include and Extend between use cases, import relationships between packages, state transitions, and others.

Saying that a subclass “knows” its superclass may seem an inadequate anthropomorphism to some readers. However, anthropomorphism is a very common rhetorical device in software engineering and computer science literature. The fundamental principle of information hiding is a good example of this. In his seminal paper on the subject [23], Parnas states that the aim of modular programming is to “allow one module to be written with little knowledge of the code in another module”. Following this tradition, we use the rather informal expression “knows about” to convey the more technical meaning “depends on”. Whenever an element simply mentions (references or points to) another element, a dependency is induced, since a change in the mentioned element will possibly affect the mentioning element. Therefore, the subclass “knows” its superclass in this sense of dependency -- in the same sense that the principle of information hiding seeks to minimize.

Dependency is then a crucial concept in this research. So is dependency defined in the UML specification [21] p. 61: “A dependency is a relationship that signifies that a single or a set of model elements requires other model elements for their specification or implementation. This means that the complete semantics of the depending elements is either semantically or structurally dependent on the definition of the supplier element(s).”

---

1 We quote page numbers of Superstructure version 2.3 (May 2010). Version 2.4.1 (August 2011) does not introduce any substantial modifications regarding our research subject. Version 2.5 (not formally released yet, last beta version is October 2012), even if it merges the Infrastructure and Superstructure documents, is only a minor revision to UML 2.4.1.
Summing up, we can say that in object oriented programming every generalization induces a dependency, a one-way knowledge, from the subclass to the superclass. Indeed, in every dependency relationship, (a) the dependent element requires the presence of the independent element, and (b) changes in the independent element may affect the dependent element: dependency means “knowledge” or “awareness”. The generalization as such is not a dependency in UML, since the dependency is a different relationship with its own notation and characteristics, but we can say that a generalization induces a dependency [6]. Minimizing dependencies between implementation artifacts (in this case, programming classes) is a general objective of software development, as well as avoiding mutual or circular dependencies, so that a clear definition of dependencies between these artifacts is crucially important [9].

However, when we consider other different contexts where the generalization relationship is used, we find that it is still an asymmetrical relationship (i.e. superclass and subclass necessarily play different roles), but not necessarily characterized by a one-way knowledge or dependency. If UML is to be used in many different contexts, it should be flexible enough and avoid to universally impose a unique notion of generalization. Equally, as we will see, other directed relationships are asymmetrical without implying one-way knowledge. Many critics of UML have argued that its present definition is neither precise nor understandable [22], that it has no exact meaning or is too inflexible [11]. If understandability is a key characteristic of models for pragmatic reasons [24], the same can be required of metamodels to be useful and effective. Unfortunately, the UML standard omits much of the rationale and historical background that is required for a full understanding of its subtleties, even though knowing the history of how the metamodel evolved would be most useful to understand it [25].

UML models can be used to represent different kinds of realities, such as software systems or “real world” systems [8] [11]. In particular, UML models can be used to represent programming code, i.e. one of the parts of the system implementation. The fact that the model can be transformed into the system by means of code generation is not against the fact that the model is also a representation of the code. In this paper we will mainly use code as the reality represented by the model, because it manifests particularly well the implications of directionality at system, model and metamodel levels.

Our purpose in this paper is to provide some conceptual clarifications that hopefully will contribute to improve the understanding of the UML metamodel regarding directed relationships, and in particular the generalization relationship. We will use the case of generalization as a benchmark to analyze the implications of asymmetry and directionality at different metamodeling levels M1 and M2, and then we will extend our conclusions to other directed relationships. Since we are going to discuss the convenience of one-way relationships at different metamodeling levels, we need to introduce the distinction between levels M0, M1 and M2 (we do not introduce M3 because it is not required for our discussion). In particular, our claim that all meta-associations should be two-way at the metamodel level (M2) does not preclude at all the use of one-way associations at the model level (M1).

The rest of the paper is structured as follows. Section 2 considers the directionality of generalization in other domains, and in particular in the context of model

---

2 OMG documents explicitly use models to represent program code, see for example Semantics of a Foundational Subset for Executable UML Models (fUML) [20], p. 17.
manipulation tools. Section 3 explains the UML metamodel of generalization, and how it permits or complicates finding the children classes of a given class, as required by model manipulation tools. Section 4 criticizes this metamodel, arguing that levels M0, M1 and M2 have perhaps not been properly distinguished. Section 5 presents the cases of other directed relationships in UML. Section 6 summarizes other related works that criticize the present definition of the UML or try to reach a deeper understanding of its structure. Finally, section 7 contains a summary of our proposal and section 8 the conclusions of the paper.

2 The directionality of generalization in different contexts and at different abstraction levels

In the wider sense of set theory, from which object oriented programming has borrowed the concept of subclassing (= subsetting), generalization can be considered adirectional in the sense that no direction is implied, or that knowledge is mutual through the relationship. That is, from a logical or semantic point of view, the generalization relationship just captures the subset relationship, not implying in itself a one-way knowledge. It is anyway a “directed” or asymmetrical relationship, in the sense that the two participants in the relationship play roles that are not interchangeable. This inherent asymmetry of the relationship is adequately depicted by using an arrow symbol (or any other kind of asymmetrical symbol).

In object oriented programming languages it is convenient to make the definitions of classes unaware of the definitions of their subclasses to support software evolution through subclassing. That is, in this particular context, as explained in the introduction, the asymmetrical character of generalization is captured by one-way dependencies between classes for the convenience of software development. However, this need not be the same in all imaginable contexts. For example, UML is often used to model “real world” problems such as biological taxonomies, where the meaning of generalization does not imply exclusive one-way knowledge. In other words, generalization is inherently an asymmetrical relationship (what UML calls a “directed” relationship), even though the practical implications of this asymmetry (one-way knowledge or not) can be different in each semantic domain where the relationship is used: one-way in object oriented programming, two-way in real world problems.

In the context of model manipulation tools, still within the realm of object oriented programming, but at a higher level of abstraction, generalization is, or should be, a two-way relationship. Obviously, the subclass must know the superclass. Why should the superclass additionally know its subclasses? Let’s see some examples.

- In a graphical modeling tool, when a superclass is deleted from a model, all generalizations having it as a target are simultaneously deleted, and maybe, though not necessarily, the subclasses are automatically marked as invalid (because the superclass they were dependent on has disappeared). Analogously, if the superclass is not deleted, but modified, the subclasses will possibly be affected: for example, adding an abstract operation to the superclass forces the concrete subclass to redefine it. This indicates that the superclass “knew” in some way those generalizations and source subclasses, so that the action of deleting or modifying a superclass can have transitive
effects on them. (More precisely speaking, it is the modeling tool that knows the subclasses of a given superclass.)

- In an Integrated Development Environment, when a superclass is deleted from the project, or modified, the user will be equally interested in marking the subclasses as invalid or requiring attention. That is, a transitive effect to the subclasses, without having to wait until the parser or the compiler detects the error, is interesting for the user. This propagation implies the existence of knowledge in the direction opposite to the one indicated by the directionality of the generalization.

- In a software reuse tool (a tool to retrieve reusable software artifacts from a repository), and analogously in ontology management tools and information retrieval tools, where the generalization-specialization relationship is also present, when a certain element is searched for, not only hyperonyms (hierarchically superordinated by generalization) may be relevant for the result of the query, but also hyponyms (hierarchically subordinated). Again, as in the previous cases, actions performed on a concept are transitively propagated to its sub-concepts.

Note that in these three cases we do not claim that the superclass should know its subclasses at the level of program code, which would contradict the essence of generalization and the objective of minimizing dependencies in object oriented programming. Modeling tools, and other analogous tools, work in a more abstract level that is not the code itself but a representation of the code. In this level it is indeed required that the element that models the superclass knows the element that models the subclass, as shown by the examples above. That is, in metamodeling terminology, we are no longer concerned about relationships between elements in the represented reality (M0), but about relationships between elements in the model (M1) that represents this reality. This distinction between an artifact and the element that models it, even though both bear the same name (“class”, “generalization”), is crucial, and we will come back to it later. We are not questioning here whether the generalization relationship should be one-way in the M0 level (i.e. the code), but whether it should also be one-way in the M1 level (i.e. the model).

Moreover, the reason to demand two-way knowledge between concepts in real world problems (M0 in a different context) is different from the reason to demand it between modeling elements (M1 in object oriented context). Two-way knowledge is possibly required to model real world concepts because generalization in a wider sense does not imply exclusive one-way knowledge. On the other hand, two-way knowledge is required between modeling elements both for practical and theoretical reasons. We first analyze the practical reasons (section 4), which could not be enough to accept a proposal of modification to the metamodel. Then we proceed on with the analysis of the theoretical reasons (section 5), which in our opinion are the true ones to accept our proposal.

The practical implications of the directionality of generalization at M1 can be posed this way: in the context of model manipulation tools, how to find the children classes of a given class. In our particular case, we are developing a service to retrieve

---

1 We are following here the interpretation of Atkinson and Kühne [1], Bézivin [2], Miller [16], and others, according to which M0 is the represented reality, not the instances of M1, as the official definitions apparently, but not very clearly, state. We refer the reader to these authors for very interesting discussions about this topic.
UML models stored anywhere in the Internet. UML diagrams can be found in the web in multiple formats: pure images or images within textual documents in common formats (JPG files, PDF files, etc.), proprietary formats for specific CASE tools, standard XML interchange formats (XMI), etc. Even though they are all essentially the same kind of documents for human eyes—they are UML models—the plurality of file formats prevents a uniform access and query system to this particular subset of information existing in the web. Therefore, the purpose of this project [26], still under development, is to build a search engine that is able to interpret a UML diagram according to its syntactical structure, regardless of the file format. To do that, the tool must be able to fully manipulate a UML model and its elements, traverse all relationships in the model, etc.

This service uses an extension of the nUML library [17] to manipulate UML 2.x models in a way that conforms to the UML specification. This library provides primitive operations to access a superclass from its subclasses, but it does not provide instead any operation to traverse the generalization in the opposite way. Of course, using the provided primitives it is possible to build operations that offer the desired result, but the fact that we did not find primitive operations to access a subclass from its superclass was a cause of perplexity. This led us to question whether nUML was deficient in its implementation of the UML metamodel, or else, on the contrary, we had to look for the deficiency directly in the metamodel. We call it “deficiency” because initially we thought the possibility to traverse the generalization in both directions at the M1 level was completely natural. We then verified that nUML adheres to the UML metamodel of generalization: it is the metamodel which does not allow this functionality. Therefore, a deeper comprehension of this metamodel was required.

3 The metamodel of generalization

The generalization relationship is metamodeled in UML [21] by means of the metaclass Generalization, connected through two meta-associations with roles called general and specific to the abstract metaclass Classifier (concrete subtypes of Classifier are Class, DataType, Interface, and so on). Generalization is a subtype of DirectedRelationship, also abstract. Classifier has also a derived reflexive meta-association with role name /general (see Figure 1).

![Figure 1](image-url) Excerpt of Figure 7.9 in UML 2.3/2.4.1: abstract syntax of generalization.
As we can observe from navigability indications, the meta-association with role name general between Generalization and Classifier is one-way; instead, the meta-association with role name specific is two-way, being the opposite role called specialization; besides, it is a composition. That is, a classifier and its outgoing generalizations do know each other, whilst a classifier does not know its incoming generalizations. This explains the impossibility to navigate from a parent classifier to its children, which is the practical problem we have identified in the previous section. The derived reflexive meta-association with role name /general, which is also one-way, is defined from these two meta-associations and the operation parents( ) in the corresponding sections Constraints and Additional Operations in the explanation of the metaclass Classifier in the UML specification ([21] p. 53; see Figure 2):

Constraints: [1] The general classifiers are the classifiers referenced by the generalization relationships.

general = self.parents( )


Classifier::parents( ): Set(Classifier);
parents = generalization.general

Therefore:

general = self.parents( ) = self.generalization.general

Figure 2. Definition of the derived reflexive meta-association with role name /general.

Note too that the roles opposite to general and /general are anonymous, though they could have been called specialization (counterpart to generalization) and /specific. This agrees with the fact that there are no additional operations to navigate these two meta-associations in the opposite direction. If these two roles had a name and the meta-associations were two-way, it would be easy to define a reciprocal operation children( ) and navigate directly from a classifier to its children (see Figure 3).

Classifier::children( ): Set(Classifier);
children = "specialization".specific

Therefore:

"specific" = self.children( ) = self."specialization".specific

Figure 3. Navigating towards the children would be easy if the additional operation children( ) were defined and the roles specialization and /specific did exist. (between quotation marks because they do not really exist).
In fact, these roles have been anonymous until version 2.3 of the UML (May 2010). Version 2.4 beta (January 2011), later consolidated as 2.4.1 (August 2011) has awkwardly modified Figure 7.9: the anonymous role opposite to `general` has been named `generalization`, not `specialization` as we suggest. This naming is certainly not a formal error, since the two meta-association ends belong to different namespaces (the non-navigable end does not belong to `Classifier` but to the anonymous meta-association). However, we consider it is a stylistic error in the sense that it makes the diagram more difficult to read. In fact, it is the only place in the whole metamodel where the same role name is repeated in two meta-associations of the same metaclass. We think the convention (non mandatory, in any case) of naming associations ends based on the classes they touch should not have been followed in this case for the sake of clarity. For the rest, version 2.4.1 introduces no other substantial modification in this respect, and UML 2.5 (not formally released yet, last beta version is October 2012) is only a minor revision to UML 2.4.1; in fact, nothing has been changed in version 2.5 regarding our research subject, even though Figure 7.9 has been merged into Figure 9.1 with other previously separated diagrams. Anyway, this is a side question that has no implications for our main argument.

Now, given the present structure of the metamodel, it is not possible to directly navigate in the explained way; instead, it is necessary to follow an indirect path to reach the children. One possible solution is to build the operation `children( )` so that it performs an iteration amongst the whole set of classifiers in the model, searching for those ones that have a given classifier as its parent (see Figure 4). This operation could be used to produce the desired result, defining the new derived role `/specific`, and then using it in navigation expressions that would be notationally compact, but actually rather inefficient due to the iteration.

```java
Classifier::children( ): Set(Classifier);
children = Classifier.allInstances( )
->select(c | c.parents( )->includes(self))
```

**Figure 4.** Iterating the whole set of classifiers in the model to find the children of a given classifier.

A second, more efficient, possible solution would be to iterate the set of generalizations in the model, presumably with fewer elements than the set of classifiers; this solution replaces the inclusion test by an equality comparison (see Figure 5).

```java
Classifier::children( ): Set(Classifier);
children = Generalization.allInstances( )
->select(g:Generalization | g.general=self)
->collect(s:Generalization | s.specific)->asSet( )
```

**Figure 5.** Iterating the set of generalizations in the model to find the children of a given classifier.

Therefore, there exists a technical solution to the problem of finding the children classes of a given class (or, in general, the children of a given classifier). This solution is notably more inefficient than navigating the meta-associations the opposite way (if
it were possible). Besides, it requires the operation `allInstances()` to be implemented by the model manipulation tool. Of course, there can be other different technical solutions. For example, in the UML2 plugin for the Eclipse Platform, each model element has an operation `getTargetDirectedRelationships()`, which retrieves the directed relationships for which the given element is a target [4]. Its implementation is based on a cached cross reference table (a hash map), which is more efficient in time than iterating the set of classifiers or the set of generalizations in the model. It is, nevertheless, still less simple than directly navigating the meta-association the opposite way.

It could be argued that the price being paid in inefficiency or complexity is reasonable, since it is a consequence of the fact that the metamodel must express the nature of the generalization relationship. Now, is it appropriate that the directional nature of generalization be expressed in the metamodel by means of one-way meta-associations? We do not think so, and we will try to explain our position in the following section.

4 A critic to the metamodel of generalization

As we have seen, the present structure of the metamodel explains the practical difficulties found, but leaves open the question to more formal considerations. Perhaps efficiency considerations do not suffice to propose a modification to the metamodel of generalization, in order to facilitate navigating from a classifier to its incoming generalizations. That is why, in this section, we are going to investigate more deeply the directional nature of generalization, and how it should be represented in the metamodel.

4.1 Connected symbols do know each other

As we have already said, at first sight it can seem natural that generalization be expressed by means of one-way meta-associations, since the directionality of generalization in object oriented programming precisely means that the parent must not know its children. Now then, if this is true at the level of represented reality (M0), that is, in the program code, it is not so clear that the same should happen at the level of the model (M1), that is, amongst the modeling elements that represent programming artifacts.

![Figure 6. What happens if superclass A is deleted or shifted?](image)
Let us see it with a simple example (see Figure 6): consider a model with a superclass A and two subclasses B and C. If we ignore for the moment that the rectangles actually mean code fragments, that is, classes in a programming language, then we can consider the rectangles as mere graphical symbols. What should happen if we deleted the rectangle that represents the superclass A? In any modeling tool, the generalization arrows would be automatically deleted. Could we assume, then, that the rectangle was not “aware” of the arrows pointing at it? On the contrary, this familiar phenomenon shows that the graphical symbols that are connected in a diagram do “know” each other, since the actions performed on one of them have transitive effects on the connected symbols. A similar case would occur if we shifted rectangle A: the arrows representing generalizations would simultaneously move to accompany the rectangle in its displacement.

What we observe when graphically manipulating generalizations in a UML model (concrete syntax) is also observed if we look at the “internal structure” (abstract syntax) of the model itself, typically represented as a tree in modeling tools. In this case, depending on the tool, the generalization often appears as an element subordinated to the corresponding subclass in the tree structure, as if the subclass were the only one that knows the generalization. This probably stems from the fact that the meta-association specific is a composition (see Figure 1). However, if we delete the superclass in the tree, the generalization also disappears in the subclass. It is true that these transitive effects can be implemented in modeling tools by means of the technique we have described above (defining operations to obtain the subclasses by iterating the whole model), but in this case we think that the iteration, and its lack of efficiency, is not reasonable. In our view, the lack of efficiency reveals a deeper theoretical problem. Modeling elements that are interconnected do know each other in the internal structure (the model itself), and the metamodel should reflect this fact. The same argumentation could be applied in the case of an Integrated Development Environment, a software reuse tool, an ontology manager, and so on, which will equally have an “internal structure” of the information they are manipulating. Note that model manipulation tools must conform to the metamodel if they want to adhere to the standard specification. So, in a certain sense, if the metamodel does not reflect mutual knowledge between interconnected elements, then the tools are being forced to be less efficient than they could be.

If we recall section 2, it is crucial to adequately distinguish between an artifact and the element that models it, so that the rules that apply to the real artifacts need not be identical to the rules that govern the behavior of the modeling elements. For example, in Figure 6 the two rectangles B and C pointing with arrows to rectangle A mean that “B and C are subclasses of A”, but this statement is valid only if it is referred to the artifacts represented by the rectangles in the drawing: the rectangles themselves, labeled with ‘B’ and ‘C’, are not “children” of the rectangle labeled with ‘A’ (the properties of B and C, considered as modeling elements, are not inherited from A; on the contrary, the three classes are equally “linguistic instances” [1] of the metaclass Class, which determines their properties).

Indeed, this is the key to the distinction between the relationship M0-M1 and the relationship M1-M2: the model (M1) represents a certain reality (M0); the metamodel (M2), instead, specifies the syntactic rules the model (M1) must conform to. In other words, models are compound linguistic expressions, and the metamodel defines the rules that allow legal combinations of modeling elements to build valid models from
the syntactic point of view. The metamodel defines the modeling language, and only indirectly has to do with the modeled reality.

4.2 Representation and conformance

The distinction between the relationships represented-by and conformant-to has been extensively dealt-with in [2]. The code and each of its elements (the represented reality in this case) is represented-by the model, and the model is conformant-to the rules expressed in the metamodel. Continuing with our previous example (see Figure 7), if we take the generalization expressed in subclass B, the keyword extends in the Java code fragment is graphically expressed by means of the arrow between the rectangles A and B (M1), which respectively represent classes A and B (M0). If we now look at the generalization arrow as such, this is conformant to the metaclass Generalization, whilst its head and its tail are respectively conformant to the meta-associations with role names general and specific (M2). We can observe here that the asymmetric directionality of generalization is already regulated in the metamodel by means of the distinction between the two meta-associations, without need of either of them being one-way. Why is it necessary that general be one-way? Even more, if we accepted general to be one-way, why should not specific be equally one-way, from Classifier towards Generalization?

Figure 7. The relationships represented-by and conformant-to between represented reality (M0), model (M1) and metamodel (M2).

Unfortunately, the UML specification gives no reason for having adopted this decision. We can consider a variety of reasons for making some meta-associations one-way instead of two-way. In general, one-way meta-associations are easier and safer to implement (no need to take care of updates on both sides), they avoid duplication of information in external storage systems (relational databases or serialized files), and so on. So, in spite of the lack of efficiency when the relationship is reversely navigated, there are some efficiency and reliability advantages for the simpler (one-way) approach.

However, these technological reasons are insufficient to justify the decisions made. If the one-way approach is generally better than the two-way approach, why not
systematically use the former for every meta-association? But, above all, even if the UML authors had performed a technological scrutiny for each one of the hundreds of meta-associations in the UML metamodel to discriminate in each case which approach was preferable, the UML metamodel should be technology independent: the navigability of meta-associations should not depend on the ease, safety, reliability and efficiency of metamodel implementation, but on the nature of the language concepts they represent. In other words, the metamodel should specify two-way or one-way navigability whenever it is conceptually required by the language being defined, and leave the implementers the decision of the best technological choice to implement it in a model manipulation tool.

Truly, our initial approach to this problem has been technological: we have identified a lack of efficiency when navigating from the parent to the children classes. But beyond the technological aspect of the question, which is indeed disputable and dependent on a huge amount of factors, we aimed at the underlying conceptual aspect. If we accept that the UML metamodel was designed to be technology independent, the only reason for the choice between one-way and two-way meta-associations should be the nature of the language concepts to be represented. And here we come back to our previous argumentation: from the linguistic point of view, connected modeling elements do know each other, even though the artifacts they represent do not know each other. The purpose of the metamodel (M2) is to describe the modeling language (M1), not the represented reality (M0). In the represented reality (the code) the superclass does not know the subclass. On the contrary, in the modeling language the superclass symbol does know the subclass symbol. Mutual knowledge implies two-way meta-associations, but this is not what we find in the metamodel. Why?

In M0, i.e. generalization in object-oriented programming languages, the parent must not know the children; in M1, i.e. generalization in UML, the parent and the children symbols do know each other. There could be the danger that the metamodel were unduly trying to represent a feature of the represented reality (M0), instead of the features of the modeling language as such (M1). If that were the case, the UML metamodel would contain a mistaken mixture of metalevels M0, M1 and M2. If the metamodel had tried to inadequately express a feature of the represented reality, it would have gone far beyond the strict necessity: the expression of syntactic rules that allow legal combinations of modeling elements.

It would not be right to say that the metamodel has no relation at all with the represented reality (i.e. the semantic domain), but this is not what we intend, either. In essence, a metamodel is the definition of the abstract syntax of a modeling language\(^4\). Even though the metamodel, properly speaking, does not define the language semantics, it does have certain semantic contents, since, at least, it collects the domain concepts of the represented reality in the names of the metaclasses and meta-relationships. In our case, the represented reality is the particular vision of object orientation invented by the UML authors, where the domain concepts (and metaclass names) are Class, Generalization, and so on. Likewise, the way to connect modeling elements expresses in a certain way properties of the represented reality (for example,

---

\(^4\) “A metamodel is a special kind of model that specifies the abstract syntax of a modeling language” [18] p. 2; see the interpretations of authors like [10] [11] [22] and other official definitions in [19] p. 7-4, and [21] p. 1. Nevertheless, there are authors for whom the metamodel also includes the definition of the notation and semantics of the language [3] [14]. We disagree with the latter, in favor of the former.
that a generalization is a relationship between two elements). That is, abstract syntax (metamodel) and represented reality are not completely independent.

We do not claim that the metamodel must not express the directional and asymmetric nature of generalization. It certainly has to. But it already achieves this goal when it expresses the source and target of the relationship by means of two different meta-associations. What we do claim is that making one of these two meta-associations one-way is superfluous and, what is worse, harmful; it is harmful because, without a theoretical justification, it can promote the misinterpretation of metalevels. Neither the interpretation of generalization (which is in any case different in each domain, see section 2) nor the convenience of implementation must impose the navigability of the meta-associations in the metamodel of generalization. The rationale for our main claim can be summarized as follows: connected graphical symbols do know each other, and the goal of a metamodel is to specify the syntactic properties of a language, ergo meta-associations must be two-way.

5 Other directed relationships in UML

There are other directed relationships in UML that are metamodeled in a similar way to generalization. We would expect a similar behavior from the point of view of graphical modeling in all of them, that is, that the graphical symbols representing them (different kinds of arrows) must “know and be known by” the elements they connect. We present them succinctly in this section.

![DirectedRelationship](image)

**Figure 8.** Excerpt of Figure 7.15 in UML 2.3/2.4.1: abstract syntax of dependency.

The most emblematic case is the dependency relationship (see Figure 8). In an analogous manner to generalization, it is metamodeled by means of the metaclass Dependency, which is a subtype of DirectedRelationship, and it is connected to the abstract metaclass NamedElement through two meta-associations called supplier and client, the former being one-way. This structure is inherited by the subtypes of dependency: Abstraction, Realization, Substitution, and Usage. The most notable difference with generalization is that a dependency can connect several clients with several suppliers, which is reflected in the multiplicities of these two meta-associations. Other differences are: the meta-association client is not a composition (it cannot be, due to its multiplicity, contrary to specific), and the role opposite to supplier is not anonymous (contrary to general). They have in common, instead, the fact of being
respectively subsets of meta-associations source and target, which are meta-associations inherited from DirectedRelationship, as explained below.

The same schema is followed in other relationships that are also subtypes of DirectedRelationship, such as Include and Extend between use cases, import relationships between packages such as PackageMerge, PackageImport, and ElementImport, and others we simply list here: ComponentRealization, ProtocolConformance, InformationFlow, TemplateBinding, and ProfileApplication. The general structure of all these relationships is inherited from the abstract metaclass DirectedRelationship (which in turn is a subtype of Relationship), connected to the abstract metaclass Element through two meta-associations called source and target (inherited through subsetting from relatedElement, see Figure 9). Note the multiplicities of source and target are 1..*; these multiplicities are preserved in the subsetting inherited meta-associations of Dependency (supplier and client), whereas they have been restricted to 1..1 in the analogous ones of Generalization (general and specific). Other subtypes of DirectedRelationship have restricted these multiplicities, too.

Note also that both meta-associations are one-way in DirectedRelationship, whereas in most cases navigability has been added to the inherited meta-associations for the opposite of source (not for the opposite of target). This is legal, according to Liskov’s substitution principle [13], since it is like adding a new structural feature to the subtypes of Element.

Figure 9. Excerpt of Figure 7.3 in UML 2.3/2.4.1: abstract syntax of directed relationships in general.

A notable exception to this general rule is the case of state transition, which is literally defined as a directed relationship (“A transition is a directed relationship between a source vertex and a target vertex” [21] p. 587). However, in spite of being defined this way, the metaclass Transition is not a subtype of DirectedRelationship in the metamodel (see Figure 10). This metaclass is connected to the abstract metaclass Vertex through two meta-associations called source and target, both of them two-way (concrete subtypes of Vertex are State, Pseudostate, and so on). That is, in this case we have a directional, asymmetric relationship in the represented reality that, nevertheless, is metamodeled by two-way meta-associations. For some unexplained reason, the authors of the UML did not consider it necessary to transfer this directionality into the meta-associations.
6 Related work

The fact that the present definition of the UML is neither precise nor understandable has been widely recognized. O’Keefe [22] gives a number of useful criteria to improve its definition. Henderson-Sellers [11], while acknowledging the benefits of the UML for software development, comparable to the introduction of high level programming languages four decades ago, still emphasizes its immaturity as a modeling language. Cook [11] criticizes its inflexibility and lack of exact meaning. Harel and Rumpe [10], as well as Mellor [11], consider that one of the big problems of the UML has been the disparity of effort devoted to define its syntax and its semantics, which relegates UML to being a mere sketching language. Miller [11] points out that the deliberate absence of a foundation is the source for many of the errors found in the UML specification. Selic, one of the strongest supporters of the UML, underlines understandability as a key characteristic any modeling language must possess [24], but regrets the fact that the UML specification omits much of the rationale and historical background that is required for a full understanding of its subtleties [25].

The interpretation of metalevel M0 as the represented reality, not the instances of metalevel M1, can be found in Atkinson and Kühne [1], Bézivin [2], Miller [16], and others. This is related with the distinction between the ontological and the linguistic dimensions in metamodeling, due to Atkinson and Kühne [1], as well as with the distinction between the relationships represented-by and conformant-to that has been deeply studied by Bézivin [2]. The consideration of a metamodel as the definition of the abstract syntax of a modeling language (not its semantics) is well rooted in several OMG definitions [18] [19] [21] and the works of Harel and Rumpe [10], Mellor [11], O’Keefe [22], and others.

In previous works, we have developed the idea that dependency means “knowledge” or “awareness” [6], we have shown the use of UML models to represent both software and “real world” objects [8], and we have reflected on the importance of paying due attention to the phenomenon of signification and representation in software modeling in order to avoid pernicious misunderstandings [7].
7 Summary of our proposal

7.1 Origin of the anomaly

The core of our theoretical argumentation is that the purpose of the UML metamodel is to specify the features of a graphical modeling language, i.e. the abstract syntax of a graphical notation. Among these features we find no reason to assume that graphical symbols that are interconnected in a diagram know each other in only one direction. The UML metamodel itself has an unstable position in this regard. Recall the previous considerations about different meta-associations that specify different kinds of directed relationships:

- Figure 7: why general is one-way, why specific is two-way.
- Figure 8: why supplier is one-way, why client is two-way.
- Figure 9, why both target and source are one-way.
- Figure 10, why both target and source are two-way.

The origin of the anomaly we think we have detected in the metamodel representation of directed relationships could be explained through the difficulties to understand the phenomenon of signification and representation. In general, a sign or symbol is “something that stands for something else” [5], so that it is easy to make the mistake of attributing the sign properties that belong only to the reality it represents, and vice versa. That is, the mistake of confusing the levels of (a) the represented reality and (b) the model made of signs that represents that reality. Once more, it is crucial to adequately distinguish between an artifact and the element that models it, between the properties of reality and the properties of the model.

Presenting an analogy that explains this idea, if we say that “Ann has two hands”, we do not mean the word ‘Ann’ has two hands; equally, if we say that the word ‘Ann’ has three letters, we do not mean that “Ann has three letters”. Similarly, the directionality of generalization between classes in the program code (M0) does not necessarily imply the directionality in meta-associations (M2) that model the representation of generalization (M1).

If program code features had been the ground for defining one-way meta-associations in the metamodel of directed relationships, this would reveal a deeper problem relative to the comprehension of the essence of a metamodel. The English grammar establishes rules to build valid sentences such as “Ann has two hands”, by means of which we can describe the world around us. But the English grammar, in and of itself, does not specify the properties of the world, only the properties of the language we use to describe it. Analogously, a metamodel is only the definition of the abstract syntax of a modeling language ([18] p. 2, [19] p. 7-4, and [21] p. 1); a metamodel is not the specification of the properties of a certain represented reality. The metamodel of generalization must specify how generalization is represented in a model; the metamodel must not specify the inherent properties of generalization in the modeled reality (be it program code or a different one). And analogously with other directed relationships.
7.2 How the UML metamodel should be

The metamodel as such must be limited to specifying, in a technology independent way, the allowed combinations of language elements, and how these elements relate to each other. The only reason for the choice between one-way and two-way meta-associations should be the nature of the language concepts to be represented. Once we have stated that modeling elements that are connected do know each other, even though the artifacts they model do not know each other, it should be clear that the specification of navigability in the meta-associations must be two-way. The UML metamodel must express the syntactic properties of the modeling language (UML), not those of the programming language(s) it is inspired in. Besides, program code is only one amongst many of the possible semantic interpretations of UML [11]. Neither the interpretation of generalization (which is anyway different in each domain) nor the convenience of implementation must impose the navigability of the meta-associations in the metamodel of generalization.

Summing up, we must not confuse the levels M0, M1 and M2. The properties of the represented reality (M0) are expressed in the model (M1), and only indirectly in the metamodel (M2). The directionality of generalization is already sufficiently represented in M2 by the fact that Generalization has two different meta-associations with Classifier. Adding navigability to these meta-associations is superfluous and, what is worse, harmful, because it can promote the misinterpretation of metalevels.

Since the line of reasoning applied to generalization can be extended to any other directed relationship, then we could say that the metamodel must not contain any one-way meta-association. Consequently, we propose to amend the UML metamodel so that all meta-associations are two-way. This is not a minor correction, since it affects the behavior of model manipulation tools: the guiding element to develop these tools is the abstract syntax defined by the metamodel of UML. Even though the lack of efficiency in traversing the generalization relationship was the starting point of our analysis, our proposal is not based on efficiency considerations, which are relative and must be balanced against other technological considerations such as simplicity, safety and reliability. Our proposal is based on solid theoretical grounds about the nature of graphical modeling languages and the proper distinction between modeling metalevels, between an artifact and the element that models it, between the properties of reality and the properties of the model. The metamodel should specify two-way or one-way navigability whenever it is conceptually required by the language being defined. From the linguistic point of view, connected modeling elements do know each other; therefore the mutual knowledge implied in the connection must be metamodelled by two-way meta-associations. In contrast, what we find is that the metamodel lacks a clear and well justified guiding principle to assign navigability indicators to meta-associations.

Our proposal does not necessarily affect other languages that are not oriented to graphical representation. A sequential-text modeling language, for example, might not imply mutual knowledge between adjacent terms. However, any graphical modeling language could receive a similar analysis and the conclusion would be the same.

The practical and conceptual difficulties we have examined cannot be overcome by adding to OCL a direct possibility to navigate one-way relationships in the opposite direction: if it were so, the distinction between one-way and two-way navigability
would disappear, so what would be the use of indicating it? The navigability of meta-associations in the UML metamodel deserves a better justification.

8 Conclusions

Navigability means knowledge, i.e. the possibility of instances to access other instances through the links that connect them [6]. Association navigability is a useful modeling concept that should be employed when necessary, either in user models at level M1 or in metamodels at level M2. Our general warning against the use of one-way navigability for meta-associations in metamodels of graphical modeling languages does not preclude at all its proper use for associations (without “meta-”) at the user model level.

We think our proposal is not unsubstantial or unnecessary: it means a real improvement of the definition of UML [22] that will help to fulfill better the role of UML in different contexts. To make it more useful and effective, we need a simpler definition of UML, avoiding elements that are not fully justified. Speaking of the design of Algol 68 in his 1980 ACM Turing Award Lecture, C.A.R. Hoare said: “I gave desperate warnings against the obscurity, the complexity, and the overambition of the new design, but my warnings went unheeded. I conclude that there are two ways of constructing a software design: One way is to make it so simple that there are obviously no deficiencies and the other way is to make it so complicated that there are no obvious deficiencies. The first method is far more difficult” [12].

We have been rather bold in denouncing what we think is a flaw in the metamodel, maybe rooted in a misunderstanding of metamodeling levels. In any case, this is not the first time that the UML metamodel comes under scrutiny or even strong criticism, and corrections to it have been proposed [11]. In any case we urge the reader to be both skeptical and open-minded about our arguments before making a decision.

Acknowledgements

This research is supported through the Spanish Ministerio de Ciencia y Tecnología, Project TIN2007-67153, “SEMSE: SEmantic Metadata SEarch” (“Desarrollo de un sistema de recuperación conceptual mediante niveles semánticos en la representación de esquemas de metadatos”).

References


