Appeared in Science of Computer Programming, Vol. 44, Issue 1, July 2002, 5--22

Profiles in a Strict Metamodeling Framework

Colin Atkinson and Thomas Kühne

AG Component Engineering,
University of Kaiserslautern,
67653 Kaiserslautern, Germany
{atkinson,kuehne}@informatik.uni-kl.de

Abstract

If the UML is to continue to meet the expectations of its ever-growing user community it is essential that it offer a simple and coherent mechanism for users to tailor the language to their specific needs. However, current UML extension approaches are not only unnecessarily limited in the capabilities that they provide, but also break some of the fundamental tenets of meta-modeling in a multi-level framework. In particular, they are all based on the assumption that instantiation, in one form or another, is the only mechanism by which end users can apply predefined model elements in their own applications. In this paper we identify the problems associated with this limitation and explain why inheritance is also important for allowing users to apply predefined model elements. We point out the fundamental differences and relationships between instantiation and inheritance for defining UML profiles and provide guidelines as to which mechanism should be used under which circumstances. We conclude by describing why both mechanisms should be utilized in the definition of UML profiles in the context of strict, linear metamodeling frameworks.

Keywords

metamodeling, strictness, profiles, UML, inheritance, classification

Introduction

The success of the UML in a wide range of application domains has made it important to view the standard more as a family of languages, sharing a common core, than as a single language supporting minimal context-specific extensions. Current plans for the UML's evolution therefore envisage a significant shrinkage of the UML core, coupled with the definition of an enhanced extension mechanism to support the addition of domain and user specific modeling concepts [8]. Several different extension mechanisms have been proposed [5], [6], but the most prominent is the "profile mechanism" first described in a white paper for the OMG Analysis and Design Platform Task Force [10], and subsequently elaborated in later versions of the UML [9].

Like any description language, the purpose of the UML is to define a coherent and useful set of concepts that users can apply in their own work. With object-oriented languages such as the UML that offer user-definable classes, there are two basic ways of achieving this. Concepts can either be defined within the core language along the lines of traditional non-object-oriented languages, or they can be defined as classes within the predefined "standard" libraries. Both approaches are put to full use in object-oriented languages such as Smalltalk and Java which have relatively small core language definitions with comparatively large supporting libraries. The only difference between classes in these standard libraries and classes added by the user is that they are *predefined* alongside the language definition as part of the technology standard. Thus we use the adjective "predefined" to refer to any concept defined within the language standard ([9] in the case of the UML) or in the standard working environment of a particular user. Note that the notion of something being "predefined" is relative to the user. For a UML user working for a specific company, the predefined modeling concepts are those in the UML standard plus those in any profile(s) whose use is mandated by the company (e.g. the company profile etc.).

Although the currently proposed UML tailoring mechanisms differ in their details, they all take the view that the definition of predefined model elements is a matter for the "meta" level in the OMG's standard four-layer modeling architecture [9], and that users should apply the predefined model elements only by instantiating them.

In this paper we argue that the assumption that predefined model elements should only reside at the meta level is flawed, and that inheritance at the regular "model" level is also an important mechanism for applying predefined model elements. In other words, it must be possible to predefine model elements at the model level (for inheritance) as well as at the meta level (for instantiation). Using an example, we demonstrate how, for specific purposes, inheritance enables the predefinition of modeling elements and/or their properties in a much more natural way than instantiating elements from the meta level (e.g. stereotyping). After establishing guidelines as to when to use which mechanism, we apply these principles to determine how a profile mechanism should fit into a strict metamodeling architecture, such as that envisaged for an improved UML infrastructure [8].

Profiles and the Standard Modeling Architecture

All UML modeling takes place within the context of the standard four-level OMG model architecture depicted in Fig. 1.

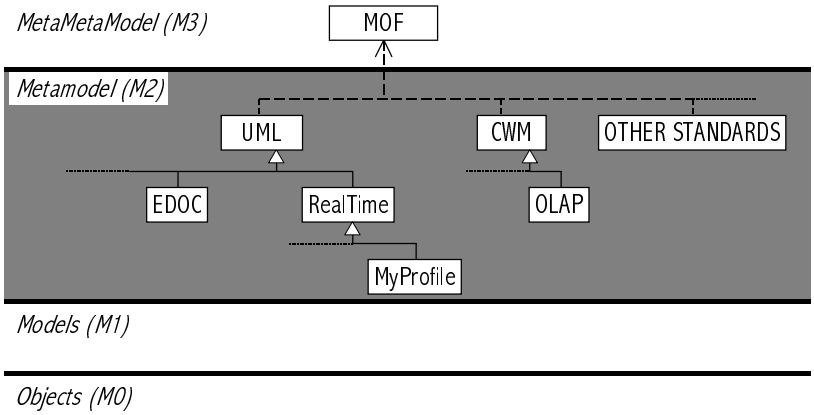

Fig. 1. The OMG view of profiles

The top (M_3) level in this scheme is the so called meta-metamodel, or meta-object facility (MOF), that defines the basic concepts from which specific metamodels are created at the meta (M_2) level. This includes the UML metamodel and other data representation standards such as the CWM, which as illustrated in Fig. 1, are regarded as being instances-of the MOF, residing at the M_2 level. Normal user models, created using the concepts of the UML or CWM, are regarded as residing at the M_1 level, and the run-time data is regarded as residing at the M_0 level.

In defining the four level model architecture, the UML specification [9] states that "A model is an instance of a metamodel" and "A metamodel is an instance of a meta-metamodel." This makes it clear that the basic relationship between the layers is intended to be the instance-of relationship. However, this definition leaves many questions unanswered, in particular –

1. What is the precise nature of the instance-of relationship?
2. What does the instance-of relationship between models mean in terms of the relationships between the model elements within the models?

The answers to these questions have a critical bearing on the semantics and practical properties of any profile mechanism. We discuss these further in the following subsections.

Instance-Of Relationship

An instance-of relationship exists between two model elements when one element, X, is instantiated from another, Y. X is then said to be an instance-of Y. Establishing an instance-of relationship can be understood both in terms of set membership and as a mechanism for deriving one model element from another.

From a set theory perspective, the instance-of relationship could more precisely be referred to as “member-of”. If X is an instance of Y, the definition of Y, known as the intension of the concept, defines the properties which all instances (e.g., X) of the concept have to satisfy. The set of all instances (i.e. members) of a concept is known as its extension.

As a mechanism for deriving one model element from another, instantiation can be understood as a creation activity, which uses a template (Y) to stamp out an instance (e.g., X). With this interpretation, the instance-of relationship could more precisely be referred to as “created-by”. When an element X is created from another Y, the attributes of Y become slots of X, with appropriate values, and the associations involving Y become links involving X. This means that the properties defined in a model element can only affect its instances, and not the instances of its instances. In other words, it can pass information across only one instantiation step. An alternative form of (deep) instantiation, in which properties can be propagated across more than one instance-of relationship, is described in [4]. However, in this paper we rely on the traditional shallow semantics of instantiation, which to date has been assumed for the UML metamodeling architecture.

If the extensions of all classes in a system are mutually disjoint, these two interpretations of the instance-of relationship are equivalent, since then the only way for an element to be a member-of the extension of a class is to be created from it. However, when the extensions of classes are allowed to overlap, and subsets of their extensions are defined, the distinction between the two interpretations becomes more subtle. Subsets of extensions are created when subtyping (or specialization as it is known in the UML) is used. If Z is a subtype of Y, the extension of Z is a subset of the extension of class Y, and every member of Z is also a member of Y. This is related to the idea of polymorphism in object-oriented systems in which an instance-of Z can be viewed as an instance-of Y, and in fact as an instance-of all Z’s superclasses.

Two important conventions of contemporary object-development approaches govern the relationship between the two views of instantiation mentioned above in the presence of subtyping. The first is that an object can be created by one and only one class. In UML terms, an object can have only one classifier. In contrast, an object can be a member-of multiple classes in addition to the one from which it is created. The other convention is that if an object X is a member-of a class Y, then either X is created by Y, or the class that creates X is a subtype of Y. In the first case X is said to be a *direct* instance-of Y while in the second it is said to be an *indirect* instance. Other strategies for establishing conformance between objects exist, but the one described above is the approach used in most statically typed object-oriented languages and the UML.

The basic goal of the meta-modeling approach described in this paper is to extend the conventional semantics of two-level object-oriented modeling to a multi-level framework. We therefore assume a metamodeling approach which is faithful to these two basic tenets of object-oriented development: namely that an object can have only one classifier (i.e., participate in only one direct instance-of relationship) and that for an object to be a member-of a class other than its classifier (i.e. an indirect instance-of), this class must be a supertype of its classifier.

Instance-of Relationship between Levels

Once the nature of the instance-of relationship between two abstract entities has been clarified, the next question is how it relates to model levels in the multi-level model hierarchy. There are two basic schools of thought on this issue, which can be characterized as "strict-" versus "loose metamodeling".

Strict Metamodeling

Strict metamodeling [3] is based on the tenet that if a model A is an instance-of another model B then every element of A is an instance-of some element in B. In other words, it interprets the instance-of relationship at the granularity of individual model elements. This can be captured in the form of a class diagram[1] as illustrated in Fig. 2. The doctrine of strict metamodeling thus holds that the instance-of relationship, and *only* the instance-of relationship, crosses metalevel boundaries, and that every instance-of relationship must cross exactly one metalevel boundary to an immediately adjacent level. This can be captured concisely by the following rule –

Strict Metamodeling: *In an n-level modeling architecture, M_0, M_1 … M_{n-1}, every element of an M_m-level model must be an instance-of exactly one element of an M_{m+1}-level model, for all m < n-1, and any relationship other than the instance-of relationship between two elements X and Y implies that level(X)=level(Y).*

This definition deliberately rules out the top level in a hierarchy of levels, since a common approach to terminate the hierarchy of metalevels is to model the top level so that its elements can be viewed as instance-of elements in the same level. In terms of the model-level "instance-of" relationship, this is described as a model being an instance of itself.

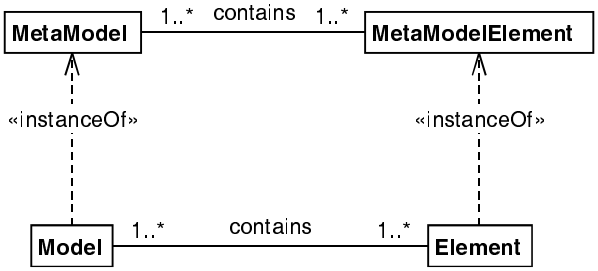

Fig. 2. Strict Metamodeling

In essence, the strict metamodeling approach simply seeks to extend the time-honored class/object distinction and instantiation semantics from classic object-oriented development to all levels in a multi-level modeling architecture.

[1] This diagrammatic representation of strict metamodeling is due to Cris Kobryn.

Loose Metamodeling

Loose metamodeling essentially encompasses all approaches which claim that one model is an "instance of" another model, but where the strict requirements on the instance-of relationship between individual model elements does not hold as defined above. In practice, this means that the location of model elements is not determined by their place in the instance-of hierarchy, but instead by other criteria. In other words, in a loose metamodeling hierarchy one simple places model elements in the model where one finds a need to mention them. Although this makes the initial definition of metamodels much easier, it also gives rise to some subtle, but significant problems.

The first problem is the blurring of the level-boundaries that arises when the contents of models are chosen solely from a utilitarian perspective. An immediate consequence of this blurring is that all kinds of relationships have to cross the boundary between metalevels, including inheritance relationships, associations and links. This in turn impacts upon the integrity of the model levels, which effectively end up playing the role of packages that only serve to group elements into subgroups of like purpose. This is not a bad thing in itself, since the value of grouping related model elements within packages has long been established. However, wrapping up what essentially amounts to an application of packages in all the baggage and paraphernalia of "meta" modeling not only becomes confusing, but is also directly misleading. Why characterize the relationship between model levels as the instance-of relationship, when, if loose metamodeling is employed, the instance-of relationship is not even the most common form of relationship between the levels?

A second and more significant problem is the need to deviate from the well-established mechanism of instantiation in object-oriented approaches to make loose metamodeling work. An example, which exemplifies this problem is the problem of defining a prototypical concept (such as the prototypical class instance, `Object`) which serves to convey upon entities the basic property of being an object. We call this the "Prototypical Concept Problem". The approach used in the specification of the UML (based on loose metamodeling) is to define the prototypical concept at the same level as the class from which it is instantiated. The model elements `Class` and `Object` both appear within the (M_2) metamodel, and are related by an unnamed association. But this requires that –

1. a modeling element at the M_0 level must be an instance-of an M_2 element.
2. a modeling element at the M_0 level must be a direct instance-of two classes (known as the "ambiguous classification" problem [4]).

This is clear in the work of Alhir [1], and [2] which have to resort to double, direct instance-of relationships when attempting to fully characterize the relationships between modeling elements within the context of loose metamodeling.

Profiles and Predefined Model Elements

Although the instance-of relationship, as elaborated above, is claimed to be the criterion identifying the model levels in the UML standard, in fact a different unstated

principle is actually used. The implicit principle is that all "predefined" concepts in the UML standard automatically reside at the meta (M_2) level, and that everything user-defined automatically resides at the model level (M_1). Thus, something is chosen to be at the metalevel because it is predefined, not because of its naturally location in the instance-of hierarchy.

To see that this is the case it is necessary to examine the current profile mechanism in more detail. In the current version of the UML [9] a profile is viewed as -

> "…a package that constitutes the definition of a UML extension. It contains a collection of Stereotypes, TagDefintions, Constraints, Comments and standard ModelElements".

In other words, it represents a set of applications of the built-in UML extension mechanisms which collectively provide a coherent set of new modeling concepts for a specific domain or application. The semantics of a profile are therefore derived from the semantics of the primitive extension mechanism upon which they are based—namely stereotypes, tagged values and constraints. Of these, the stereotype concept is the most fundamental in terms of creating new modeling concepts. Stereotypes were introduced into the UML as a way for users to logically extend the metamodel without tools having to physically change the metamodel. Thus, the stereotype concept –

> "…provides a way of branding (marking) elements so that they behave in some respects as if they were instances of a new virtual metamodel construct" [9].

This is reinforced by the idea that stereotypes behave as classifiers for stereotyped elements.

> "All model elements classified by one or more particular stereotypes receive these values and constraints…" [9].

The tagged value and constraint mechanisms do not provide a way of introducing new modeling concepts as such, but rather define additional properties of existing or newly introduced constructs. Tagged values simply provide a shorthand way of defining new meta-attributes and assigning values to them, while constraints simply define additional rules by which model elements can be utilized. Thus, apart from actually extending the metamodel itself, stereotypes represent the only mechanism for defining new model elements, whether separately or as part of a profile. The current approach used in the definition of the UML standard is to place the core (i.e. fundamental) model elements in the meta (M_2) level, and to add additional "predefined" profiles for specialized domains. Note that the very concept of predefined stereotypes is something of an oxymoron, since the original motivation for stereotypes was to provide a simple user extension mechanism. By definition, predefined model elements are not defined by individual users.

The currently predefined (or standard) profiles are the UML profile for Software Development Processes and the UML profile for Business Modeling [9]. Although it is nowhere explicitly stated in the UML standard, the message is that predefined elements must reside logically at the meta (M_2) level however they may be physically represented. Fig. 1 is an adaptation of an OMG diagram [10] of the profile concept which clearly indicates that all tailoring of the UML for specific applications is expected to take place at the M_2 level. This assumption is true also for the other proposed UML extension mechanisms [5], [6].

The UML's preoccupation with meta-level (M_2) modeling as the only way to provide a predefined set of concepts upon which users can base their work is actually somewhat surprising, since as mentioned previously object technology has a well established and successful mechanism for providing predefined building blocks – the inheritance mechanism. Object-oriented programming languages, such as Smalltalk, Eiffel, and Java feature a whole hierarchy of predefined classes, rooted in a class called `Object` from which all other classes either explicitly or implicitly inherit. Note that this predefined "Object" class is *not* a meta concept residing at the M_2 level, but is purposely provided at the M_1 level.

We believe that many of the current problems with the UML standard and the proposed profiling mechanisms stem from a failure to recognize the importance of M_1-level inheritance[2] as a mechanism for providing predefined modeling elements. Before discussing how proper utilization of this mechanism can aid in a clean definition of the profile mechanism, we first investigate, in the following section, the difference between inheritance and instantiation.

Inheritance versus Instantiation

In order to compare instantiation to inheritance as a mechanism for applying predefined modeling elements we use the well-known Observer pattern [7]. Since the UML has no generally accepted notation to depict the class of an M_1-level class (i.e. the metaclass from which a class is instantiated), we use the stereotype notation, with the understanding that this form is normally intended only for indicating instantiation from user-defined modeling elements.

Predefining a Subject Role

The Observer pattern identifies a subject role whose task it is to notify a set of attached observers whenever the subject's state changes. The observers then in turn query the subject about its state in order to synchronize their own state (e.g., a rendered view of the subject's contents). Fig. 3 shows that a subject role may attach and detach multiple observers. Whenever the subject's state changes it will call its own notify method, causing an update message to be sent to each attached observer instance. Fig. 3 also shows that the subject and observer roles are actually performed by concrete subclasses. Concrete observers have an association to a concrete subject so that they can exploit a particular interface to inquire about the subject's state (e.g., `getState()`).

This pattern is common enough to be found within the predefined class libraries of common object-oriented languages. For example, the Java package, `java.util`, defines two interfaces `Observer` and `Observable` with methods similar to the corresponding classes in Fig. 3. The question we wish to address in this paper is how can one best support the predefinition of these roles within the UML? As an example,

---

[2] Inheritance at the M_2 level also plays an important role in the extension mechanism, and will continue to do so. It is the use of inheritance at the M_1 level (or lack thereof) which is the issue here.

suppose that we wanted to apply the Observer pattern to the visualization of, say, a data table object (e.g. for displaying multiple diagram types of the same data). As explained in the previous sections, as well as using inheritance to derive a user specific version of `Subject` from the predefined abstract definition, as in Fig. 3, it is also possible to use instantiation.

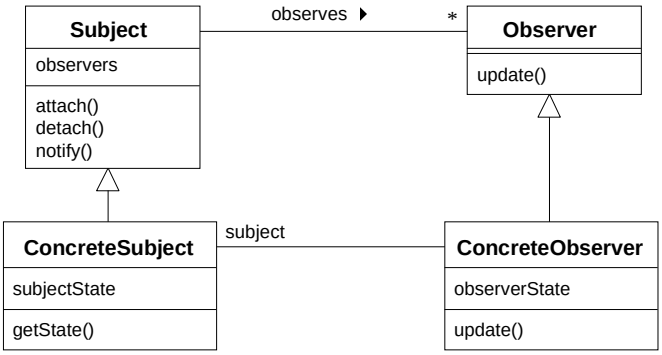

Fig. 3. The structure of the Observer pattern

Note that our goal here is not to necessarily present the best realization of the Observer pattern or to attempt to find its optimal representation using the UML. Instead, our goal is to compare the mechanisms of instantiation and inheritance for deriving user-specific model elements from predefined ones.

Subject as a Predefined M_2 Element

When using instantiation to derive a concrete subject class, the predefined version must logically appear at the M_2 level. Using stereotypes this can be achieved by introducing a stereotype named "Subject" which is used to mark classes intended to play the role of concrete subjects (see Fig. 4)[3]. However, since a stereotype can not equip the class it is applied to with attributes, class `Table` has to explicitly define the `observers` and the `notify()` method in addition to its internal state (`cells`) and inquiry methods (`getState()`).

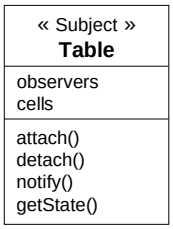

Fig. 4. Subject modeled with a stereotype

[3] Recall that a stereotype applied to a class at the M_1 level defines a virtual metaclass at the M_2 level.

The fact that concrete subjects such as `Table` have to explicitly define all their attributes and operations is a fundamental consequence of the properties of instantiation, not of the choice to support it by stereotypes. Any attributes or operations defined for an M_2-level class become slots and class level operations of its instances, and therefore can have no effect on the objects created by a further instantiation step.

Subject as a Predefined M_1 Element

As illustrated in Fig. 5, the inheritance mechanism allows concrete subjects, such as `Table` to be defined without having to explicitly list all their subject-related features. These features, instead, are automatically attained by the normal semantics of object-oriented inheritance. Consequently, this is the approach typically used in the published definitions of patterns, such as those in [7].

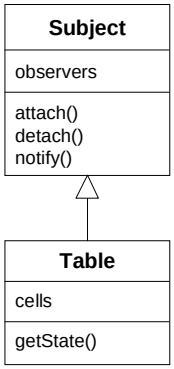

Fig. 5. Subject modeled with inheritance

If the subject role is modeled as a class at the M_1 level, a `Table` class may inherit from it, receiving all its features. Note that if the subject class only defines abstract features then class `Table` only receives constraints (i.e., the obligation to implement the abstract features). If, however, class `Subject` defines concrete attributes and methods then class `Table` is able to fulfill a subject role by only providing a specific `getState()` method. The rest is predefined by class `Subject`.

Comparing the Mechanisms

In both variants above (see Fig. 4 & Fig. 5), `Table` is classified as being a subject. However, when we used stereotyping for classification (Fig. 4), the structure of table instances cannot be influenced directly in the predefined description of `Subject`. The most that could be specified here without resorting to the definition of constraints (e.g., with OCL), is class related information such as "author" or "version" information. Stereotyping class `Table` with `Subject` actually means that a virtual metaclass `Subject` is derived from metaclass `Class` and then `Table` is instantiated from it (see Fig. 6 (a)). Thus, any attributes specified in `Subject` become class-level attributes of `Table`.

When inheritance is used, however, (i.e., class `Subject` resides at the M_1 level, see Fig. 6 (b)) one can straightforwardly predefine features, associations, invariants, etc. in class `Subject` to be received by class `Table`. In effect, the "jump" across the metalevel border has already been performed by class `Subject`, thus allowing it to predefine properties for `Table` at the same modeling level. Interestingly, the two approaches both use instantiation and inheritance (derivation) but in reverse order:

- the first *derives* `Subject` and then *instantiates* it to `Table`, whereas
- the second *instantiates* `Subject` and then *derives* `Table` from it.

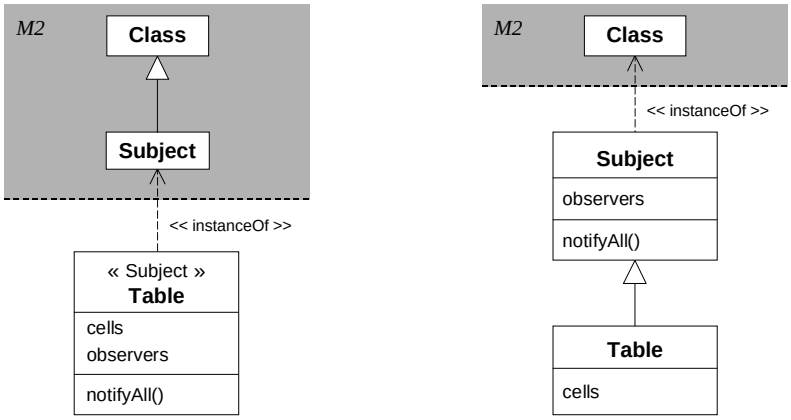

Fig. 6. Instantiation versus Inheritance

The only apparent difference is that in the latter case a link between the classes `Table` and `Subject` is established to denote inheritance. The fact that `Table` then does not have to provide subject related features is just a consequence of this link. This observation seems to suggest that predefining elements through stereotyping can, to a certain extent, be used interchangeably with predefining element through M_1-level inheritance. However, clearly the practical effects of the two mechanisms are different:

- instantiation does not affect the structure of the new M_1 elements. It is, therefore, optimally used to express non-code related information (e.g., project relevant information) or to capture implementation details, which have no effect on the stereotyped classes but on other classes (e.g., marker interfaces, such as "`Serializable`" which are only used to signal this property to other classes which actually implement serialization).
- inheritance may shape a new M_1 element through predefined constraints, interfaces, features, etc. It obviates the need for writing constraints within stereotypes, which check that instantiated M_1-level elements obey a certain structure (e.g., provide a certain attribute). With inheritance this attribute (or an association to another class, or corresponding methods) can be directly specified.

As a general observation, inheriting from M_1-level elements seems to considerably reduce the need for constraints. In the above example, the stereotype `Subject` is likely to contain a constraint, checking that the stereotyped element actually features an `observers` attribute. This property, in contrast, is guaranteed by construction when inheritance is used for the classification of subjects.

A Unifying Notation

The different orders of instantiation and inheritance observed above suggests that the name compartment of classes would benefit from a suitably defined notation that –

- highlights this phenomenon, and
- allows quick recognition of the situation at hand.

A notation commonly used to express subtyping is the "<" symbol. Hence with ":" denoting instantiation as usual, one obtains:

`Table` `: (Subject < Class)`	(Table stereotyped with Subject)
`Table` `< (Subject : Class)`	(Table inheriting from Subject)

The first line reads "First metaclass `Subject` is derived from metaclass `Class` and then class `Table` is instantiated from it", whereas the second line reads "First class `Subject` is instantiated from class `Class` and then class `Table` is derived from it". The difference between the two is captured graphically in Fig. 7. This uses a 3D variant of the Venn diagram notation in which inheritance is represented in the form of a sub-circle at the same level, while instantiation is represented in the form of a raised sub-circle. Thus the vertical level of a circle represents its location in the instantiation hierarchy, with the bottom level corresponding to M_2, the second level to M_1 and the top level to M_0.

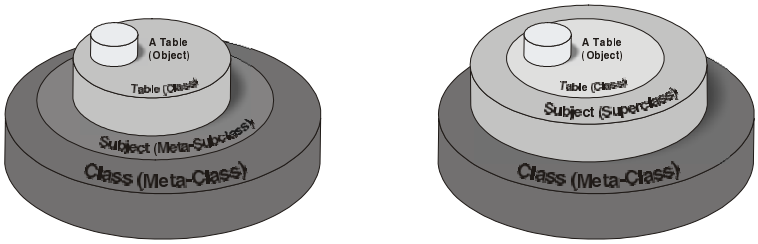

Fig. 7 Inheritance at the M_2 level versus inheritance at the M_1 level

When the stereotype syntax is used to denote instantiation and when stereotypes prefix their elements the two lines from above become:

`«Subject < Class»` **`Table`**	abbreviates to:	`«Subject»` **`Table`**
`Table` `< «Class» Subject`	abbreviates to:	**`Table`** `< Subject`

In this version, the guillemots nicely enclose all elements at the M_2 level (i.e., the gray parts in Fig. 6 and the dark-gray parts in Fig. 7).

In a further evolution one may write `Subject > Table` (instead of `Table < Subject`) and for the sake of conformance with the existing stereotype notation

even write `<Subject> Table`. Note that the ">" operator between `Subject` and `Table` still points in the right direction. Now denoting instantiation and inheritance reads:

`«<Class> Subject» Table`	abbreviates to:	`«Subject» Table`
`<«Class» Subject> Table`	abbreviates to:	`<Subject> Table`

The respective distance of the new M_1 element `Table` from `Subject` in terms of metalevels, is nicely depicted by the guillemots (« », 2 levels) and the new subtype notation (< >, 1 level). In particular, in the abbreviated forms on the right (which could be used as usual within the name compartment of a class icon) it can readily be seen that `Table` is an instance-of a metaclass in the first line above, and that it is derived from a class in the second line above.

Clearly, there is already a graphical means to express that one element is derived from another one (namely the generalization arrow). However, such redundancy already has a precedent in the UML. For example, there are three ways to express instantiation in the UML:

1. Two names separated by a colon.
2. A dependency arrow stereotyped with instance-of.
3. The stereotype notation.

Although these are strictly speaking redundant notations, each variant has an intended application context where it communicates a particular variation of instantiation. Likewise, we believe that the above proposed notation for deriving elements could specifically communicate that inheritance is used to obtain predefined modeling properties, whereas the graphical notation is typically employed to express a generalization relationship between elements within a domain.

Strict Profiles

Having discussed the subtle differences between introducing new modeling concepts at the M_2 level (for instantiation) or at the M_1 level (for specialization), we are now in a position to describe how we believe UML profiles should be defined in the context of a strict metamodeling framework. Fig. 8 gives a more faithful rendering (in comparison to Fig. 1) of how profiles are located in the four-layer metamodeling architecture. As a mechanism for predefining a modeling environment, we believe that a profile in general should contain elements at *both* the M_2 *and* M_1 levels. In other words, profiles should conceptually span modeling levels, i.e., not be confined to one modeling level as is currently the case.

Although Fig. 8 does not give the organized impression of Fig. 1, it is simply the result of taking the doctrine of strict metamodeling seriously, given that M_1 elements constitute an important part in a profile's definition. Fig. 9 gives a more detailed view of how the contents of profiles (depicted by the gray rectangle labeled L_3) are distributed over metalevels. Note that the boxes now depict individual classes while in Fig. 8 they depict profiles. Another view, that more clearly emphasizes the levels in the four-layer metamodeling architecture, is provided by Fig. 10. In this figure, corresponding shades of gray belong to the same profile.

Predefined ≠ Meta

The basic goal of a profile is to define a set of modeling elements, which users in a specific domain can apply to their own application. Thus, from the perspective of an individual user of the UML, a profile defines the set of predefined modeling elements that he/she can use as the basis of his/her own modeling work.

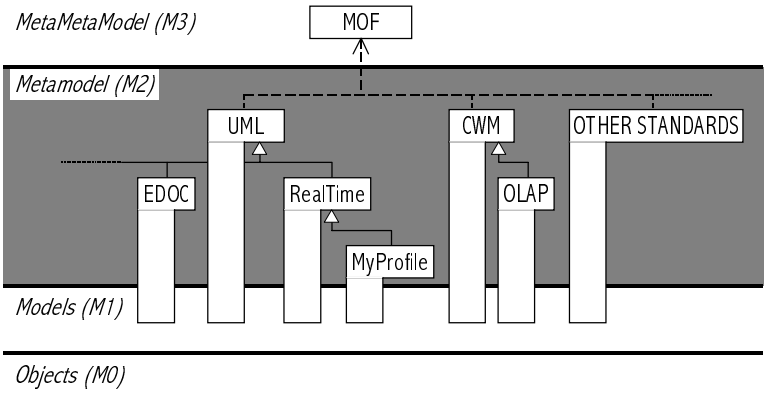

Fig. 8. Profiles containing M_2 and M_1 elements

This includes the so called "root profile" (labeled L_2 in Fig. 9) which defines the standard set of predefined elements that are part of the UML's core specification.

The key difference between the new way of defining profiles proposed in this paper, and the approach described in the existing literature, is that a profile is no longer restricted to just one level in the metamodeling hierarchy. On the contrary, profiles (including the root profile) will typically consist of elements at both the M_2 and M_1 levels. In principle it is also possible for a profile to contain predefined elements at the M0 level. For example, the "constant" objects in Smalltalk (e.g. integers, characters, Boolean values), could be viewed as special predefined objects at the M_0 level. However, we do not expect this to be common in practical UML modeling scenarios.

Rather than arbitrarily allocate model elements to levels based on whether or not they are "predefined" or "user defined", the model elements in a profile are allocated to metalevels according to their logical place in the "instance-of" hierarchy. This reflects the fundamental observation that definition time (i.e. being predefined) and level occupancy (i.e., being at a particular metalevel) are two completely different concerns. In a nutshell: "predefined ≠ meta".

As illustrated in Fig. 10, therefore, profiles generally cut across model levels in the four-layer metamodeling architecture (one profile corresponds to one particular shade of gray). For example, the root profile, which defines the UML core, consists of regular metamodel elements at the M_2 level, and several model elements at the M_1 level. Typical users of the UML core will therefore add their own classes at the M_1 level as *instances* of the predefined M_2 elements, but also as *specializations* of the predefined M_1-level elements. Advanced users who wish to define a new profile, can add new elements at both the M_1 and M_2 levels as specializations of existing modeling

elements at those levels. In this way, it is possible to build up a hierarchy of profiles, each adding to the set of predefined modeling elements in previously defined profiles by specialization at *both* the M_2 and M_1 levels.

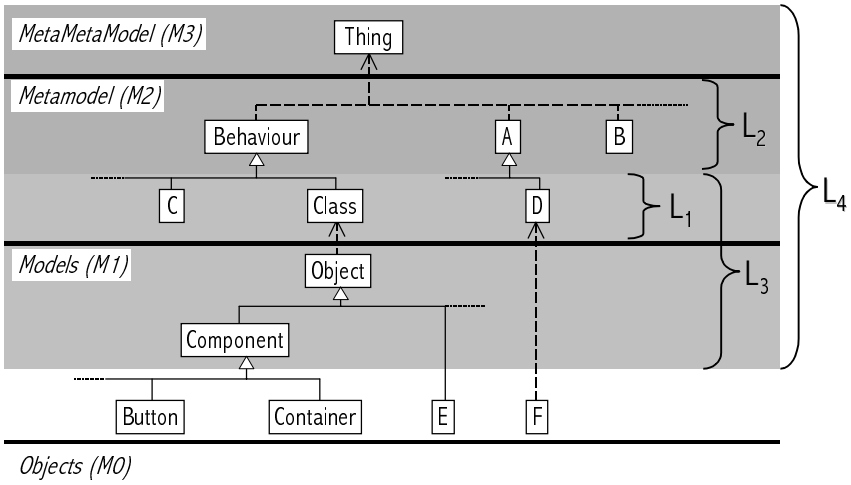

Fig. 9. Naming the modeling layers

The Prototypical Concept Problem

The model of profiles depicted in Fig. 9 illustrates how the "predefined ≠ meta" principle helps solve the prototypical concept problem outlined at the beginning of the paper in a way that is consistent with the principles of strict metamodeling. Instead of forcing prototypical concepts, such as the class `Object` and the class `Link` to reside at the M_2 level, these classes are instead allowed to reside at the M_1 level, which represents their natural location as far as the instance-of hierarchy is concerned.

Defining `Object` at the M_1 level and defining all user M_1-level classes as specializations of `Object`, (either directly or indirectly by inheriting from an already existing M_1 element) removes the "ambiguous classification" problem mentioned in section "Profiles and the Standard Modeling Architecture".

Note that any M_0 entity is still a direct instance-of some M_1 entity (which in turn is an instance-of the M_2 entity `Class`) *and* also an indirect instance-of `Object`. Since every M_1 entity (directly or indirectly) specializes `Object`, every M_1 instance (i.e. an M_0-level entity) can be regarded as an (indirect) `Object` instance. In this way M_0-level entities receive all properties of being an object without requiring them to be a *direct* instance-of two entities at the same time. In Fig. 10, therefore, the single M_1-level class within the UML core profile would correspond to the prototypical class `Object`. Note that this is an established approach in many object-oriented language models, such as Smalltalk, Eiffel and Java, where all classes have a common `Object` class as their root ancestor.

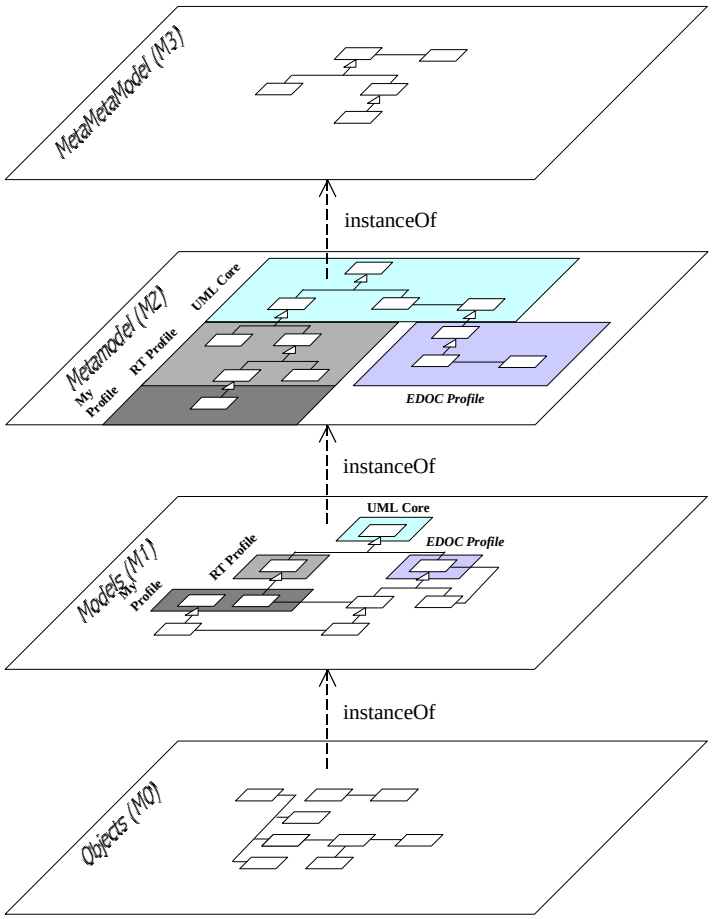

Fig. 10. Predefined entities at the M_2 and M_1 levels

This approach not only has the advantage that the class `Object` has its natural place in the multi-level metamodeling-architecture, thus avoiding the problems that arise when contravening the rules of strict metamodeling, but it also allows instances of user defined classes to be automatically endowed with a predefined set of attributes and methods.

Conclusion

With the envisaged shrinkage of the UML core, and the growing emphasis on user tailorability, the quality and flexibility of the profile (i.e. extension) mechanism will play an increasingly critical role in the UML's future success. This is reflected in the

level of interest in UML extensibility, and the growing set of proposals for the next version of the UML extension mechanism. However, as pointed out in this paper, the current set of proposals are based on an implicit, but fundamentally flawed, assumption that tailoring of the UML is a matter only for the UML metamodel (M_2) level. This assumption is not only invalid, but as explained in the paper, is at odds with long established principles of object modeling. Strict metamodeling offers the best opportunity to place future versions of the language specification on a sound footing and, hence, is envisaged in current plans for the UML's evolution [8].

The definition of a profiling mechanism that is consistent with the rules of strict metamodeling (a so called *strict profile*) requires model elements to be allocated to metalevels according to their natural location in the instance-of hierarchy rather than whether or not they are predefined from the perspective of a user. The result is an approach to UML extension, which uses regular M_1-level inheritance as well as instantiation to enable users to build upon a predefined set of building blocks. The predefined building blocks, therefore, are distributed across multiple levels in the metamodeling architecture, rather than being concentrated at one specific (M_2) level in the metamodeling hierarchy. Distributing predefining elements among multiple levels in this way not only avoids the semantic distortions that are necessary to support the "predefined ≡ meta" principle implicit in current approaches, but also facilitates a more natural allocation of properties to user classes and objects according to the mechanisms discussed in this paper. By directly shaping the structure of M_1 elements through the use of M_1-level inheritance, the need to use a constraint language in order to check a desired structure is avoided in many places.

The strict profiling principles outlined in this paper are essentially independent of the notation used to define, instantiate, or specialize individual modeling elements. For example the approaches described by Cook et al. [5] and D'Souza et al. [6] are both compatible with—and could be use to embody—the notion of strict profiles. Nevertheless, the practical application of the approach would greatly benefit from appropriate notational support that applies the concepts of instantiation and specialization in a level independent way. The UML currently supports two main notations for instantiation, one between the M_1 and M_0 levels (regular class instantiation) and one between the M_2 and M_1 levels (stereotyping). This paper provides suggestions for unifying the two approaches together with a shorthand notation for inheritance. When supported by an appropriate notation, we believe that the notion of a strict profile outlined in this paper will help form the basis for the infrastructure of the next version of the UML.

References

[1] S. S. Alhir: Extending the Unified Modeling Language. At: home.earthlink.net/~salhir, 1999

[2] J. Álvarez, A. Evans and P. Sammut: MML and the Metamodel Architecture, Workshop on Transformations in UML (WTUML'01), associated with ETAPS'01, Genova, Italy (2001)

[3] C. Atkinson: Supporting and Applying the UML Conceptual Framework, in: Proc. of UML'98 (1998)

[4] C. Atkinson and T. Kühne: The Essence of Multi-Level Metamodeling, in: Proc. of UML'01 (2001)
[5] S. Cook, A. Kleppe, R. Mitchell, B. Rumpe, J. Warmer and A. C. Wills: Defining UML Family Members Using Prefaces, in: Proc. of TOOLS 32, (IEEE 1999)
[6] D. D'Souza, A. Sane and A. Birchenough: First Class Extensibility for UML – Packaging of Profiles, Stereotypes, Patterns, in: Proc. of UML'99 (1999) 265–277
[7] E. Gamma, R. Helm, R. E. Johnson and J. Vlissides: Design Patterns: Elements of Reusable Object-Oriented Software. (Addison-Wesley, 1994)
[8] C. Kobryn: UML 2001: A Standardization Odyssey. Communications of the ACM (42):10 (1999) 29–37
[9] OMG: Unified Modeling Language Specification, Version 1.4. OMG document ad/01-02-13), 2001
[10] OMG: White Paper on the profile mechanism, OMG Document ad/99-04-07, 1999