Systematic Stereotype Usage

Colin Atkinson[1], Thomas Kühne[2], and Brian Henderson-Sellers[3]

[1] University of Mannheim, Mannheim, Germany
`colin.atkinson@ieee.org`
[2] Darmstadt University of Technology, Darmstadt, Germany
`kuehne@informatik.tu-darmstadt.de`
[3] University of Technology, Sydney, Australia
`brian@it.uts.edu.au`

Abstract. As one of the UML's main extension mechanisms, stereotypes play a crucial role in the UML's ability to serve a wide and growing base of users. However, the precise meaning of stereotypes and their intended mode of use has never been entirely clear and has even generated much debate among experts. Two basic ways of using UML stereotypes have been observed in practice: one to support the classification of classes as a means of emulating metamodel extensions, the other to support the classification of objects as a means of assigning them certain properties. In this paper we analyze these two recognized stereotype usage scenarios and explain the rationale for explicitly identifying a third form of usage scenario. We propose some notational concepts which could be used to explicitly distinguish the three usage scenarios and provide heuristics as to when each should be used. Finally, we conclude by proposing enhancements to the UML which could support all three forms cleanly and concisely.

Keywords UML extension mechanism, stereotypes, classification, transitive properties.

1 Introduction

In the context of object-oriented modeling, the notion of stereotypes was first introduced by Rebecca Wirfs-Brock [1] as a means of classifying objects according to their responsibilities within a system. In other words, in addition to classifying objects by a natural domain specific type (e.g., ATMKeypad), her idea was to classify objects according to the way in which they contribute to the system's implementation, such as "interface object" or "coordinator object" etc. The categories for this *secondary* classification are drawn from the roles the objects play within the system (see Fig. 1) and are thus design/implementation-oriented. Consequently, while names of natural types (i.e., class names) would typically be derived from the problem domain the names of stereotypes would typically be derived from the solution structure. Thus, stereotypes can essentially be understood as a way of "branding" objects of a particular primary (natural)

type with additional design-focused roles or properties. Since Wirfs-Brock used stereotypes primarily for explanation purposes, she did not map the stereotype concept to a specific mechanism such as multiple inheritance.

This ability to add additional classification information to a concept in a concise and straightforward way made Wirfs-Brock's stereotype mechanism ideal for the designers of the UML who were looking for a way of allowing users to add additional modeling concepts to the UML without having to extend the metamodel directly. Since the introduction of the UML, therefore, stereotypes have primarily become known as the "lightweight"

Fig. 1. Secondary Classification

extension mechanism of the UML, allowing users to tailor the language to their own needs without having to directly deal with the UML metamodel. The choice of stereotypes as a (restricted) language extension mechanism was not only motivated by usability concerns, however, but also as a concession to many tool vendors who at the time were unable to support direct metamodel specialization.

In the sense that it allows model elements to be "branded" with additional classification information, the stereotype mechanism defined in the UML standard [2] is entirely faithful to the original concept of Wirfs-Brock. However, the "official" UML extension mechanism differs from the original usage of stereotypes in one very important way. Whereas Wirfs-Brock used stereotypes to brand *objects*, the UML standard defines stereotypes as a way of branding *classes*. In the terminology of the UML's four level model hierarchy, the original stereotype mechanism was used to brand elements at the instance level, while the UML standard presents stereotypes as a means for branding elements at the type level.

In the previous paragraph, we were very careful to refer to the "official" *definition* within the UML standard, because in practice many UML users have applied the UML stereotype mechanism not in the way defined by the UML standard but rather in the way intended by Wirfs-Brock, that is, as a way to brand instances. Thus, in practice one finds that users of the UML use stereotypes in both styles—to brand types and to brand instances. It seems that the the notational economy of stereotypes makes them an attractive alternative to the traditional (M_1-level) inheritance mechanism as a way of branding an object, even when the latter captures the real-world scenario more accurately. One can therefore observe in the literature a large number of stereotypes applied in a very loose style. The result has been an intense debate among researchers and experts about how stereotypes should and should not be used, e.g., whether the original Wirfs-Brock usage should be considered valid in the context of UML [3].

It is overly simplistic, however, to assume that the two aforementioned basic usage scenarios span the full space of options. On closer analysis, it turns out
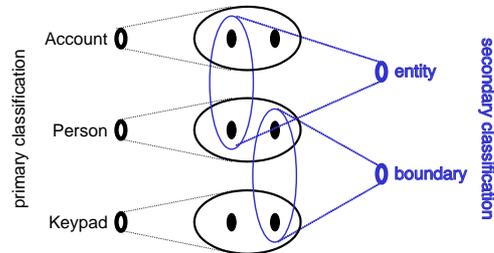
that many real world scenarios are not naturally represented by either of these usage styles, but that a third interpretation that simultaneously brands a class and all its objects is the most appropriate. In other words, often, properties of real world concepts naturally imply both forms of classification at the same time.

Our goal in this paper is to investigate this phenomenon by fully characterizing the three forms of stereotype usage scenarios and describing a strategy for working out which form should be used for a particular real-world scenario. In the following section, we first analyze and give examples of the two basic usage scenarios and then elaborate on the third kind. We subsequently equip modelers with a litmus test to identify which kind of stereotype they are facing, provide guidelines as to when a certain form should be used, propose notational measures to make the intent of a particular stereotype explicit, and finally outline how the three kinds could be most naturally accommodated in a clean multilevel approach.

2 Generally Recognized Stereotype Usage Scenarios

In this section, we illuminate and discuss the two main types of stereotype usage scenarios that have been recognized and debated in the literature [3]. Then, we introduce the third form, which—although not explicitly recognized as such in the literature to date—is possibly one of most common real world modeling scenarios.

When discussing how stereotypes are used in practice, particularly when analyzing examples of stereotype applications, it is important to bear in mind that the original intent of the modeler can never be known with absolute certainty. There is always an element of uncertainty in our interpretations of what the modeler originally had in mind. This can be regarded as being both a strength and a weakness of stereotypes in the UML. It is a strength because it means that users are able to exploit stereotypes without getting bogged down in detailed semantic issues, but it is a weakness because if the modeler had a clear meaning in mind this knowledge is lost. Consequently, in this and the following section, when we classify stereotype usage scenarios we do so on the basis of our interpretation of the real world situation that the modeler is trying to capture.

2.1 Type Classification

The UML 1.4 specification states that:

> A stereotype is, in effect, a new class of metamodel element that is introduced at modeling time [2] (page 3-33).

Hence, the motivation for the use of stereotypes is essentially to extend the standard modeling vocabulary offered by the UML by facilitating the additional classification of types. The mechanism by which this is achieved within UML 1.4 is described as—

> ... a way of defining virtual subtypes of UML metaclasses with new metaattributes and additional semantics [2] (page 2-79).

«commentedClass»
Worker
{commenter = Bob}

Fig. 2. Official stereotype use

In other words, a new type of modeling element becomes available that users can make use of at the M_1 level. According to this definition, with the exception of "redefining stereotypes" [4], stereotyped elements behave as if they were instances of the base modeling element. Thus, stereotyping of this kind provides a way of branding types so that they can be associated with special semantics. If one attaches a stereotype (e.g., «commentedClass») to a class (e.g., Worker), (see Fig. 2) for example, the effect is the creation of a (virtual) subclass of the M_2-level metaclass Class (see Fig. 3). The stereotyped class (Worker) is then considered to be an instance of this virtual subclass.
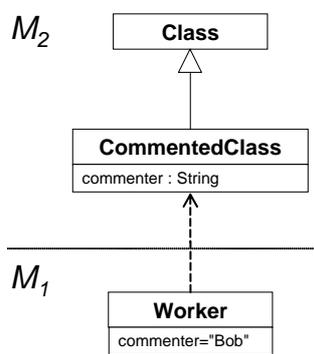
M_2

Class

CommentedClass
commenter : String

M_1

Worker
commenter="Bob"

Fig. 3. Effect of stereotype on metamodel

Thus, in our example, the stereotype «commentedClass» brands Worker rather than its instances. This becomes most evident when considering stereotypes such as «abstract» or «interface» which unambiguously refer to the classes they stereotype, since there are no direct instances of abstract classes or interfaces. The UML standard contains several examples of this "official" use of stereotypes amongst the so called "standard elements" [2]. Standard Elements are predefined constraints, stereotypes and tagged values that augment the UML metamodel with additional "out of the box" concepts. For instance, the UML has a standard stereotype «framework» that is used to brand packages that contains model elements specifying a reusable architecture for all or part of a system. Similarly Fontoura et al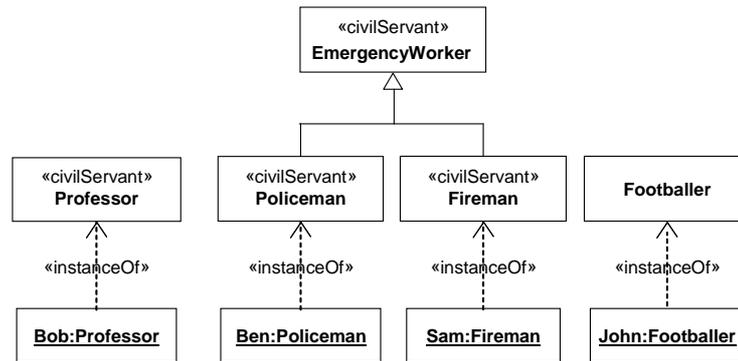. define stereotypes «application» and «framework» which brand elements as belonging to a specific application or to a generic framework [5]. Clearly, the properties of an instance at run-time are not influenced by whether their classifier originated from generic framework artifacts or was introduced as part of the application-specific customization of the framework.

Tagged Values Tagged values are intended to be used in tandem with stereotypes as a means of assigning values to user types. They are a way to supply additional information along with the branding (see Fig. 2, specifying the commenter). Tag definitions correspond to (meta-)attributes, defining what type of value can be assigned to a type, and tagged values actually define the value (see Fig. 3). Thus, tag definitions essentially correspond to attributes of (virtual) metaclasses, while tagged values correspond to slots of user types. From version 1.4 of the UML, tag definitions should be used in conjunction with stereotypes only, which reinforces the model of a stereotype as defining a virtual metaclass, and a stereotyped element as representing an M_1-level type.

Fig. 4. Instance Classification with Stereotypes

2.2 Instance Classification

The previous section described the "official" purpose of stereotypes as defined in the UML standard. As such, this has been characterized by some authors as the "correct" way of using stereotypes. However, there are numerous practical examples of stereotype usage in the UML literature where the only purpose of the stereotype is to classify objects rather than to classify classes. When used in this way, stereotypes effectively serve as a shorthand for regular inheritance between classifiers at the M_1 level. For example, Conallen uses «Applet» to stereotype a class OnlineGame [6] (page 169). It seems obvious that instances of OnlineGame should be considered to be applets, whereas OnlineGame itself is simply an ordinary class. An alternative way to obtain the assumed modeling scenario would therefore have been to make OnlineGame a subclass of an M_1-level superclass Applet. Note that the JAVA AWT library does exactly this and provides Applet as a class from which to inherit. After all, only the instances, the executable applets, need to have certain features, which is why these are defined by their classes (or superclasses). The classes themselves are not distinguishable from other classes in any reasonable way, though. Therefore, the intent of stereotyping a class as an «Applet» is to brand the instances rather than the class itself.

The use of stereotypes for instance classification is also commonly found where there is a desire to brand objects as playing a particular role in a system. Design pattern roles (e.g., Observer) played by domain classes are a frequent example, but more application-specific examples also abound.

Fig. 4 illustrates a small example from the domain of pay role management. In order to pay the appropriate salary to workers, taking into account tax deductions and pension schemes, it is necessary to keep a model of the different kinds of professions and which individuals belong to which professions. Fig. 4 illustrates four different kinds of professions and models four different individuals belonging to each of the professions. It also indicates that three of the professions also convey the status of being a "civil servant" on the individuals serving in that profession. In Germany this is a particularly important distinction since different tax and pension rules apply to civil servants with respect to other wor-
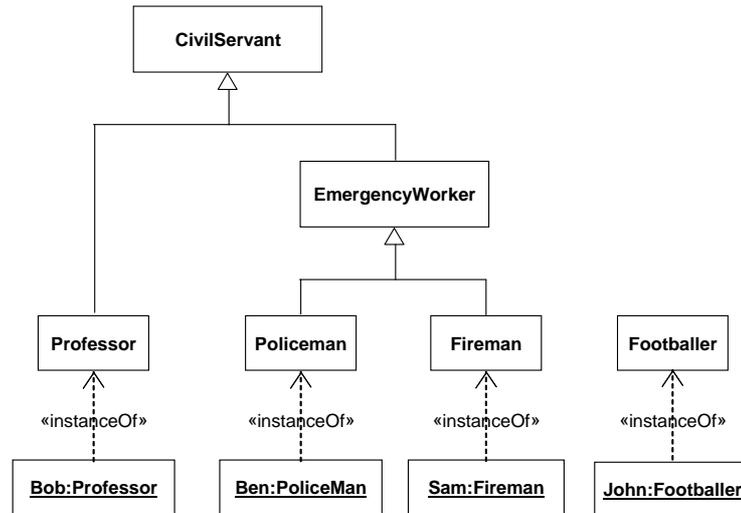
Fig. 5. Instance Classification using Superclasses

kers. Thus, the diagram explicitly captures the fact that three of the professions constitute civil servants while one (Footballer) does not.

In this example, the ≪civilServant≫ stereotype is essentially being used to define an additional property which certain workers exhibit within German society. Similar uses of stereotypes can be found in many domain-specific UML models. However, when used in this style the stereotype is essentially serving as a short hand way of defining an inheritance relationship.

The domain knowledge captured by Fig. 4 is also captured in Fig. 5 but using regular inheritance rather than stereotypes. In fact, it is the more appropriate version, since the individuals (such as Bob, Ben, etc.) are civil servants and not the professions (such as Professor, PoliceMan, etc.). If a shorthand way of expressing such a modeling scenario is to be used then—instead of using stereotypes—one could use angled brackets (e.g., <CivilServant>) instead of guillemets (≪civilServant≫) in the class compartment as proposed in [7].

Yet, without literally interpreting the UML standard users merely assume that they can (wrongly) use stereotypes as depicted in Fig. 4. It is easy to see, though, that such attempts to capture the nature of instances simply cannot work [8], since instantiation is not transitive. To illustrate this, let's assume that the stereotype ≪civilServant≫ defines a tagged value "pension" to be associated with stereotyped elements with values "normal", "super", etc. In our example (see Fig. 4), the tagged value (e.g., "{pension = normal}") would be attached to Professor and not to Bob. Such class—as opposed to instance—features, only make sense if one wants to introduce profession types, such as BlueCollarProfession and WhiteCollarProfession. These could be modeled as stereotypes to be used for Professor etc. Useful tagged values (e.g., entryLevelQualificationRequired) could then be defined and would correctly apply to the classes and not to the instances.

It is sometimes argued that traditional instance classification by using inheritance is not powerful enough to express certain advanced semantic properties [9]. For instance, it is argued that the quality of being a clock (e.g., an active notion of ticking) could only be captured with a stereotype so that a tool could generate appropriate code accordingly. This special instance behavior cannot be achieved by using a superclass Clock. However, this is an artificial result of the different levels of support given to stereotypes and subtyping by UML tools. Stereotypes do not inherently provide any better support for capturing such semantics. If a tool can exploit the existence of a certain stereotype to attach special semantics to modeling elements, it could just as well exploit the fact that the model element is a subtype of a certain special supertype (e.g., Clock).

We conclude this section by observing that even some of the "standard elements" stereotypes defined in the UML specification may at first sight also be interpreted as serving to support instance classification. Take the ≪parameter≫ stereotype, for example. This is applied to associations to brand them as being of a temporary nature. However, the property of "temporariness" clearly applies primarily to the instances of associations—the links—rather than to the associations themselves. However, it seems that the association itself may also be distinguished from other purely structural, non temporary, associations. The next section puts the spotlight on exactly this issue.

2.3   Transitive Classification

In the previous two sections, we identified two commonly observed ways of using stereotypes in practice: one "official" way that conforms to the UML standard (i.e., type classification, such as ≪interface≫) and one "unofficial" way that does not conform to the standard way (i.e., instance classification, such as ≪civilServant≫) and is thus strictly speaking incorrect. In line with the current literature we have until now implied that these are the only two options.

However, as hinted at by the ≪parameter≫ stereotype this is something of an over simplification. In describing the ≪parameter≫ stereotype, we pointed out that it is primarily intended as a way of branding instances, but also left open the question as to whether its also makes sense to think of the type (the association itself) as being branded by this stereotype.

In general, we make the following observations about the problem of mapping real world scenarios to the classification forms identified in the previous sections:

- There is no published systematic procedure to work out whether type classification or instance classification is the best match.
- Often both forms of classification apply.

The ≪parameter≫ example is a good case in point. Clearly, this scenario includes instance classification, since it is ultimately the links which exhibit a temporal property. But it is also the case that an association stereotyped with ≪parameter≫ is distinguished from other associations. The ≪parameter≫ label is essential information for a reader of the class diagram to understand that this particular

association is not motivated by structural considerations but only indicates a temporal communication path.

Such a scenario clearly constitutes a third form of usage scenario, which we refer to as *transitive classification*. This usage scenario refers to situations in which the intent of the user is to brand both the type and the instances of the type. In other words, the user intends the branding of the type to *transitively* apply to its instances as well. Thus, in a sense this scenario represents a combination of the type and instance classification scenarios described in the previous sections.

In fact, when a modeler wishes to add a new natural type (i.e., add to the set of *primary* classifiers), the intent is usually to apply this third transitive form. When one removes secondary classification (such as «controller» or «commentedClass») from a formerly decorated element the element will essentially remain unchanged. When, however, one removes a stereotype whose purpose is to extend the set of modeling element types (such as «parameter») the natural type of the model element will change from a special kind to a standard type, involving significant changes in semantics, including completely changing the nature of the instances. Whether their type is said to be «commentedClass» or not is not relevant for instances, but whether their type is said to be, e.g., «active» certainly heavily influences their behavior.
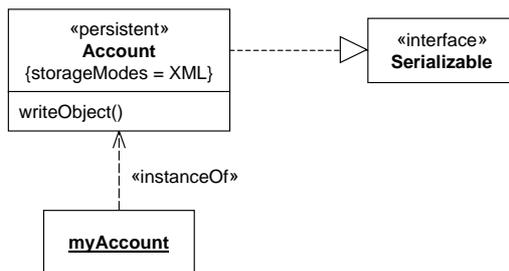

Fig. 6. Transitive Sterotype

As a typical example of "transitive classification" consider the situation in Fig. 6. It is obvious that instances of Account should be persistent and not the class Account itself. However, as well as requiring the instances to support a writeObject() operation, for the persistence mechanisms to work one must also require certain properties of class Account itself. Here, Account is required to realize the Serializable interface and also specify a format according to which its instances are to be serialized. The «persistent» stereotype as used in Fig. 6 therefore affects both type and instances. Unlike in pure type classification (e.g., stereotype «commentedClass») the intent is that distinct type properties transitively apply to instances as well.

As a second example, let us revisit the civil servant example in the previous section. In this example it is clear that objects should be characterized as "civil servants", and that classes (i.e. Professor, Policeman, etc.) are not distinguished from other classes in any reasonable way. But what if the property of being a civil servant carries with it some information that holds for all civil servant objects. For example, in many countries, civil servants have to complete a minimum number of years service to qualify for a full pension when they retired. Since this piece of information is constant for all civil servant instances, it is most naturally

defined at the class level (e.g., as a static variable in JAVA). We would then have a situation in which both classes and instances are affected by the "civil servant" property, i.e., we would have a transitive property.

Fig. 7 illustrates the idea of transitive classification in relation to the Professor example first introduced in Fig. 4 and Fig. 5. In this diagram the intent of the stereotype ≪civilServant≫ is not only to brand the class Professor itself (since it now has a distinguishing characteristic: the "minYears" tagged value which flags it as different from other types), but also to brand the professor instances as being different to normal workers. Because of the taggedValue "minYears", the inheritance mechanism used in Fig. 5 is no longer adequate for our needs, but using a stereotype only may lead to a purely official interpretation missing the point about objects carrying the "civil servant" property as well. It is important to reiterate that we are referring to the modeler's intent when identifying

Fig. 7. Transitive Sterotype

this (and the other) stereotype usage scenarios. With the current definition of stereotypes and instantiation in the UML it is not possible to syntactically distinguish transitive classification from pure type classification. Support for addressing this issue will be discussed in section 4.
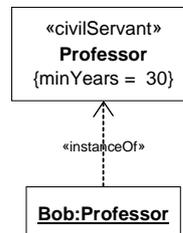
3   Classification of Stereotype Usage Scenarios

Having motivated and introduced the different kinds of stereotype usage scenarios, we now present a systematic classification of the different scenarios and describe concrete criteria for choosing between them. This classification is orthogonal to other classification approaches, such as that of Glinz et al. [4] that concentrate on categorizing usages conforming to the official interpretation of stereotypes.
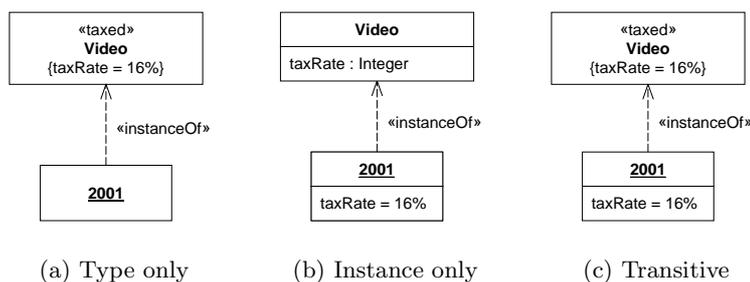

(a) Type only   (b) Instance only   (c) Transitive

Fig. 8. Forms of Classification

Fig. 8 shows a practical example of the three different cases side-by-side. In Fig. 8(a) the modeler intends to brand class Video only, adding the information that this product type has a tax rate of 16% associated with it. The tax rate information is not accessible from instances of Video. Fig. 8(b) shows the opposite extreme in which the modeler wants to classify Video objects and give them each a separate tax slot. Without any additional constraints there is nothing to control the values assigned to the tax slots of individual instances of Video. These values may or may not be the same. Fig. 8(c) shows the intermediate situation in which the modeler wants a tax rate to be visible at the level of individual objects, but wants to fix the value of the tax rate for each object at the value of the type. Thus, the intent in Fig. 8(c) is to brand instances as well as the type. Notice that with current stereotype notation it is impossible to distinguish the situations shown in Fig. 8(a) and Fig. 8(c) from the stereotyped class alone.

Table 1 summarizes the three different scenarios in tabular form. The left hand column represents type classification, in which the intent is to brand the type only, and not the instances. The middle column represents regular instance classification in which the intent in using a stereotype is only to brand the instances of a type, not the type itself. As has already been pointed out, this situation should be handled more appropriately with regular inheritance (i.e., the classifier of the instances is defined to be a subtype of a supertype corresponding to the stereotype). Finally, the right hand column of Table 1 represents the transitive classification usage scenario in which the intent is to brand the instances of the type and the type itself.

Table 1. Forms of Classification

Type Classification	Instance Classification	Transitive Classification
e.g., ≪abstract≫	e.g., ≪applet≫	e.g., ≪persistent≫
affects types only	affects instances only	affects types & instances

The difference between these three forms can also be explained in terms of their effects on the type and/or instance facets of the model element to which they are applied. As explained in [10], model elements representing types generally have two facets—a type facet which describes the properties of their instances (e.g., attributes and methods), and an instance facet which describes the element's properties as an object (e.g., tagged values). In terms of these facets—

1. type classification purely affects the *instance facet* of types,
2. instance classification affects the *type facet* of types only, and
3. transitive classification affects *both* facets of types.

With this terminology, we have an easy way to find out which category a certain stereotype belongs to. By checking which facets of a model element at the M_1

level are influenced by the stereotype, one can immediately tell which kind of stereotype one is dealing with.

The systematic view just established is perhaps also helpful in explaining why modelers use stereotypes for instance classification even though the UML definition explains them in terms of type classification. As we already mentioned, in such cases the use of inheritance at the M_1 level would be more appropriate, since inheritance is the most natural way to influence the type facet of a type. The type facet of a subtype is simply extended with that of the supertype, which is exactly what is needed when one wishes to make sure that a set of instances carry certain properties. However, it is the case that a generalization relationship is a link between two types at the M_1 level, i.e., such a link must be regarded as belonging to the instance facet of a type. Therefore, the use of a stereotype—which can only affect the instance facet of a type in the non-transitive case—seems understandable, since one can imagine that the modeler associates a designated type facet with a designated instance facet. One way to make this assumption true, i.e., truly make the stereotype have the effect of transitive classification, is to use a constraint which demands any type equipped with this stereotype to inherit from a particular class (e.g., civilServant1). Another, more cumbersome way is to use a much more complex constraint which spells out the full type facet required for "CivilServant" types.

While we have just established a way to actually make a stereotype realize transitive classification using plain UML only, we still need to be aware of the difference between a stereotype whose sole purpose is to classify instances and a stereotype which affects both type and instances, i.e., transitive classification. How to make the choice between the two cases and subsequently how to notationally distinguish them is the subject of the following two sections.

3.1 Usage Guidelines

As conceptual modeling mechanisms, with clearly distinct semantics (captured in Table 1), the difference between the three usage scenarios is easy to understand. What is much more challenging, however, is working out which scenario best fits a particular real world situation. A systematic process for making this decision is given in Fig. 9.

Instance classification is the most easy to match to real world situations, since it actually corresponds to an ordinary classification of instances, where one uses a type to describe the common properties of a set of objects. Thus, in scenarios where there is no type level information (i.e., no properties of instances are fixed at the type level nor does the type need to be distinguished from other types), instance classification is the obvious choice. In terms of the Video example, the property "price" is a good illustration (see Fig. 10), assuming the price can vary from video to video, this would be represented as an attribute at the type level, and be given different values for each instance. No information is required at the type level.

When there is some property that could reasonably be applied to the type, the situation becomes more difficult. The existence of a type-level property implies

either transitive or type classification. As specified in Fig. 9, one now needs to decide whether instances are affected or not. The difficulty in doing this arises because there is virtually no form of type-level information which cannot, in some form or another, be regarded as relevant to the instance level as well. Take the ≪commentedClass≫ stereotype, for example. Normally, one would think that software objects do not care about whether their classes are commented. But on the other hand, one cannot exclude a situation in which someone wants to reject all objects from being dynamically added to his system unless they can testify that their classes have been developed with the use of comments.

Also, in the case of the video example, should the property of the tax rate be associated with the class only, with the objects only or with both (see Fig. 8)? Assuming that the tax rate is fixed for all videos, it is reasonable to view the tax value as belonging to the type only. However, from the perspective of clients of video objects, the tax rate is just a normal property that they might be interested in (like price). The fact that it is constant for all video instances might be irrelevant at the instance level. Since there is a case for viewing it as either a type-level property or an instance-level property, there is also a case for viewing it as both (i.e., transitive property).
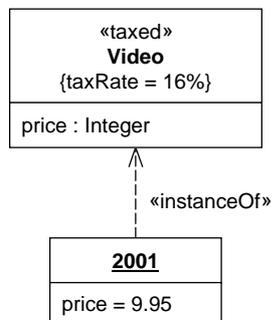

Fig. 9. Supporting the decision process

Fig. 10. Priced Object

Thus, from these examples it can be seen that a decision for one of the classification scenarios purely based on the nature of concepts in a real world scenario cannot be made. The choice heavily depends on the needs and intent of the modeler. In the case of the video example, we are assuming that the real world scenario dictates that taxes are constant for videos of a particular kind. This, therefore, implies that "taxRate" is at least a type-level property—ruling out the pure instance classification scenario. The question then is whether or not "taxRate" should also be viewed as an instance-level property. This is where the purpose of the model and the needs of the modeler come in. The fundamental question at this point is whether the modeler wants tax rates to be visible to clients of the instances. In other words, should a client of a video instance be able to directly determine (i.e., see) its tax rate? If the answer is yes, the tax property is best thought of as transitive (i.e., belonging to the type and the instances) and the transitive classification approach should be chosen. If no,
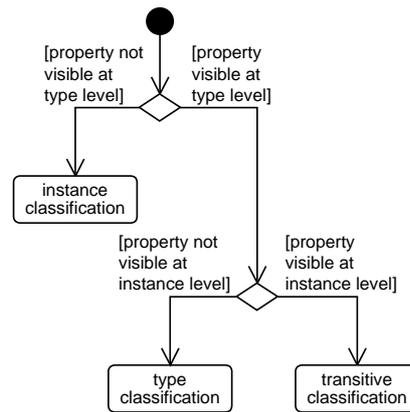
the tax property is best regarded as merely a type-level property, and the type classifications scenario should be chosen.

What remains to be discussed is how to make sure one uses a proper name for the stereotype. We now provide a litmus test for determining whether a stereotype name is appropriate or not. Assuming that the correct kind of stereotype has already been determined, and has been called "S", one question to ask is:

> Do I under any circumstance want instances of the stereotyped element to be understood as "S"-instances?

If the answer is yes, then the stereotype name is wrong. For example, if it is the case that an OnlineGame instance ought to be regarded as an "applet" instance then "Applet" is a good candidate for an Applet superclass name, but not for the stereotype. Just as class names spell out what their instances (i.e., objects) are (e.g., applets), stereotype names ought to spell out what their instances (i.e., types) are (e.g., AppletType). A stereotype name does not necessarily need to have the "-type" suffix, though. A stereotype name ≪component≫ makes sense since types branded with this stereotype are particular components, whereas their instances are usually referred to as component instances.

It cannot be overemphasized that if one wants instances to be regarded as "S"-instances then one is not really dealing with a proper stereotype name but with the name of an M_1-level supertype. What can be achieved by a stereotype ≪AppletType≫, however, is that classes like OnlineGame inherit from a superclass Applet. In this way—exploiting the terribly imprecise meaning of "is-a" [8]—not only an OnlineGame is-an applet (is-a → instance-of) but OnlineGame is-an applet (is-a → kind-of) as well, which seems to be the primary motivation for (incorrectly) using an ≪Applet≫ stereotype in such cases.

4 Notational Support

Having identified the primary stereotype usage models and discussed the heuristics that can be used to choose between them, we now discuss how the scenarios could be distinguished notationally. Modelers not only need a systematic way of obtaining the kind of stereotype most appropriately fitting the situation in hand, they also need a way to unambiguously document their decision.

In the first subsection, we discuss strategies for distinguishing them using the current version of the UML and then, in the following subsection, we discuss possible approaches that might be employed in future versions.

4.1 Current UML

Fig. 8 illustrates that, by considering class Video alone, it is not possible to deduce which of the two stereotype kinds was intended. We therefore propose slight notational enhancements to the current version of the UML so that all three kinds of stereotypes can be distinguished. Figure 11 depicts our suggestions for all three cases.
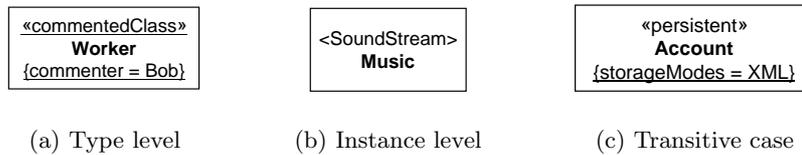
«commentedClass» **Worker** {commenter = Bob}	<SoundStream> **Music**	«persistent» **Account** {storageModes = XML}
(a) Type level	(b) Instance level	(c) Transitive case

Fig. 11. Notational support

Note the use of an adjectival stereotype name in Fig. 11(c). It is equally valid to think of both Account itself and an Account instance as supporting persistence.

Also, note the underlined tagged values which indicate that this information is meant to be useful for the type level only. Whenever a tagged value needs to be visible to instances as well (as in "taxRate = 16%") it should not be underlined.

Note, however, that while underlining tagged values works for conveying intent, i.e., restricting the visibility of a class feature (e.g., commenter) to be visible to the class only as opposed to all of its instances as well, it is not suitable to control whether a class feature (such as taxRate) shall be a part of an object's feature list (as in Fig. 8(c)). In order to achieve the level of expressiveness to independently control visibility and feature extent it is best to assume a true multi-level modeling approach for UML modeling.

4.2 Advanced Support

In [11] it is suggested that the traditional two level modeling assumption be abandoned and UML modeling be based on a multi-level approach involving user metatypes in addition to user types and user instances. Just like stereotypes, user metatypes are above the level of user types, providing a means of type classification. A semantic model for such an approach is "Higher Order Logic" [12]. For instance, types of (user-)types can be regarded as second-order predicates, expressing properties of first-order types.

With the availability of a user modeling level above the normal M_1 type level and in the presence of "deep instantiation" [11] it is easy to control where the features involved in the scenario of Fig. 10 become visible and where not.

Figure 12 illustrates how one can control the visibility of "taxRate". In Fig. 12(a) "taxRate" is just an ordinary (meta-)attribute, giving rise to a slot at type Video only. This is analogous to a tag definition/tagged value pair in the UML. In contrast, Fig. 12(b) specifies "taxRate" to be of potency two, i.e., makes it appear as an attribute (rather than a slot) at type Video. Therefore, this attribute gives rise to a corresponding slot at instance 2001.

Note the underlining of "taxRate" in Fig. 12(b), meaning that "taxRate" is a so-called dual field rather than a simple field. In contrast to simple fields, dual fields may already be assigned a value before they become a slot. This allows the value of "16%" to already be assigned a value at the type level. Furthermore, we used a potency value of 1 in "$16\%^1$" to indicate that this value is constant for

(a) Tax as a type property   (b) Tax as a transitive property
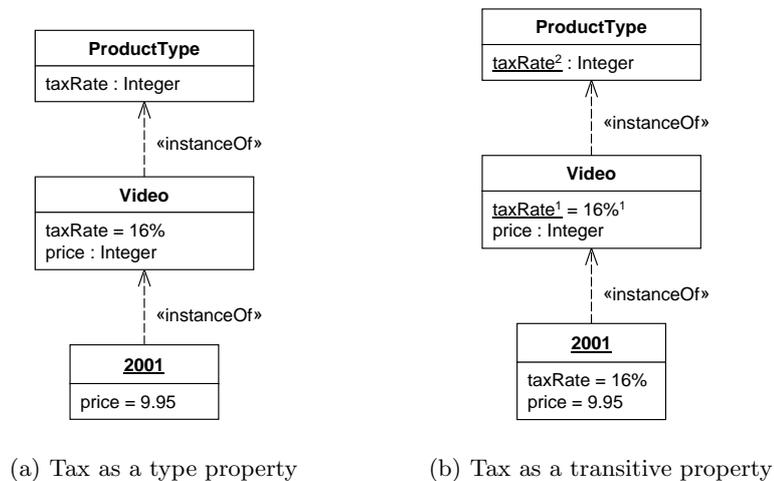
Fig. 12. Choosing the desired tax scope

the level below. Had we used "16%" instead, it would have meant that the type Video is associated with a tax rate of 16% but that any Video has a "taxRate" slot which may be assigned any individual value (analogous to the price slot).

Such a multi-level modeling approach not only makes stereotyping and tagged values redundant as mechanisms in their own right[1] but furthermore makes it trivial to control the scope of properties independently of the level where they are introduced. The modeler's intent can be captured in a concise and intuitive manner without ever requiring constraints that enforce certain type facets to be present.

5 Acknowledgements

6 Conclusion

Thanks to their notational conciseness and convenience of use, stereotypes have earned a permanent place in the affections of UML users and have become an indispensable part of the UML's extensibility mechanisms. However, as explained in this paper and others, they are often deployed by users in ways that do

---

[1] Notational shortcuts as presented in Fig. 11 are still valuable and should be made available.

not strictly conform to the UML standard and can therefore be ambiguous or misleading to readers of the model. Our goal in this paper has been to first identify and understand the different de facto ways in which stereotypes are used by UML modelers (whether they be "official" or "unofficial") and then to consolidate practical guidelines for working out which scenario makes sense under which circumstances. We then elaborated on how the different stereotype usage scenarios could be distinguished notationally, both in terms of existing UML notation and in terms of possible future notations.

We observed that a suitable interpretation of the UML standard would actually allow both type and instance classification usages for stereotypes but beyond that we have established that a simplistic reduction to two basic modes is insufficient. We identified that many real world scenarios are actually best thought of as combining these two forms so that they simultaneously classify the type as well as its instances. For this form of usage scenarios we introduce the term "transitive classification".

In fact, one of the possible reasons for the extensive debate on what stereotypes mean and how they should be used has been that the three forms have never hitherto been made fully explicit. It turns out that many examples of stereotype applications in the literature are best understood as transitive classification, rather than simply type ("official") or instance ("unofficial") classification. Since transitive classification best matches the properties of many real-world situations, it will continue to be used in the future regardless of the intended "official" stereotype semantics and regardless of how well this case can be expressed with standard stereotypes. However, in the spirit of providing a language which is as expressive and accurate as possible we believe that it is important for future versions of the UML to take all three stereotype usage scenarios into account, and to provide users with explicit notation for specifying which scenario they are intending to use.

References

[1] Rebecca Wirfs-Brock, Brian Wilkerson, and Lauren Wiener. Responsibility-driven design: Adding to your conceptual toolkit. *ROAD*, 1(2):27–34, July–August 1994.

[2] OMG. *Unified Modeling Language Specification, Version 1.4*, 2000. Version 1.4, OMG document ad00-11-01.

[3] Brian Henderson-Sellers. The use of subtypes and stereotypes in the UML model. *Journal of Database Management*, 13(2):43–50, 2002.

[4] Stefan Berner, Martin Glinz, and Stefan Joos. A classification of stereotypes for object-oriented modeling languages. In Robert France and Bernhard Rumpe, editors, *Proceedings of the 2^2 International Conference on the Unified Modeling Language*, LNCS 1723, pages 249–264, Berlin, October 1999. Springer Verlag.

[5] Marcus Fontoura, Wolfgang Pree, and Bernhard Rumpe. *The UML Profile for Framework Architectures*. Addison-Wesley, 2002.

[6] Jim Conallen. *Building Web Applications with UML*. Addison-Wesley, Reading, MA, 2000.

[7] Colin Atkinson and Thomas Kühne. Strict profiles: Why and how. In Andy Evans, Stuart Kent, and Bran Selic, editors, *Proceedings of the 3rd International*

Conference on the UML 2000, York, UK, pages 309–323, LNCS 1939, October 2000. Springer Verlag.

[8] Colin Atkinson, Thomas Kühne, and Brian Henderson-Sellers. To meta or not to meta – that is the question. *Journal of Object-Oriented Programming*, 13(8):32–35, December 2000.

[9] Bran Selic. Re: Ask about UML. email communication 10:15:56 -0400, September 2001. email to third author and others.

[10] Colin Atkinson and Thomas Kühne. Meta-level independent modeling. In *International Workshop Model Engineering (in Conjunction with ECOOP'2000)*. Cannes, France, June 2000.

[11] Colin Atkinson and Thomas Kühne. The essence of multilevel metamodeling. In Martin Gogolla and Cris Kobryn, editors, *Proceedings of the 4th International Conference on the UML 2000, Toronto, Canada*, LNCS 2185, pages 19–33. Springer Verlag, October 2001.

[12] D. Leivant. Higher order logic. In D. M. Gabbay, C. J. Hogger, and J. A. Robinson, editors, *Handbook of Logic in Artificial Intelligence and Logic Programming*, volume 2, pages 229–321. Oxford Science Publications, 1994.