Connecting Powertypes and Stereotypes

Brian Henderson-Sellers, University of Technology, Sydney
Cesar Gonzalez-Perez, University of Technology, Sydney

Abstract

Powertypes constitute an advanced OO modelling mechanism that is usually utilized in the form of a specific pattern. Stereotypes comprise the basic customization and extension mechanism in UML, and are also used following a certain pattern. Although different in purpose, these two patterns present some interesting similarities and are shown here to become structurally identical in specific circumstances. This fact can help reduce the apparent complexity of UML and may be of special importance for tools that store and transform models that use powertypes and stereotypes.

1   INTRODUCTION

The concept of powertype has been discussed in the literature and adopted in UML 1.x as a peripheral and advanced modelling mechanism. Stereotypes, on the other hand, are prominent in UML. In this paper, we investigate what turns out to be a very close connection between these two OO modelling constructs, defining the patterns in which they are often used. When an additional (often implicit) class is introduced into the stereotype pattern, the result is easily seen to be a powertype pattern. This leads potentially to a revision of the way in which instantiation and subtyping relationships are defined in the context of a strict multilevel metamodelling framework such as the one used by UML and the OMG.

2   POWERTYPES

Powertypes were introduced into OO modelling by Odell (1994) and are often considered as an advanced modelling technique. A *powertype* is a type the instances of which are subtypes of another type (called the *partitioned type*). Powertype and partitioned type are thus related indirectly through the entities that are instances of the former and, at the same time, subtypes of the latter. This indirect relationship is often modelled as an explicit, direct relationship (see Figure 1 for an oft-used real world example). Also,

because instances of the powertype *are* subtypes of the partitioned type, they are objects as well as classes concurrently. UML mentions powertypes but does not offer any notational support for the concept of an entity that is both an object and a class. Consequently, this paper will represent such entities as an individual object and an individual class inside a greyed ellipse (see Figure 1).
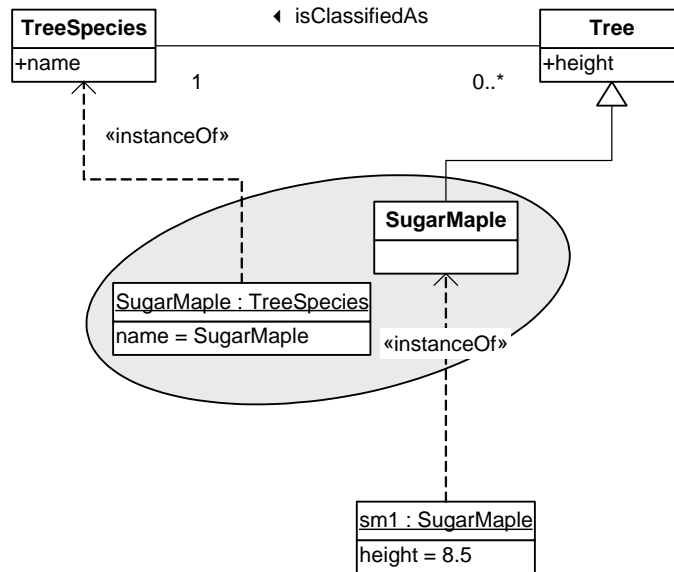
Figure 1. TreeSpecies is a powertype of Tree, and Tree is a type partitioned by TreeSpecies. SugarMaple is both a type and an object. A grey ellipse is used to denote this since UML does not offer a notation for a model element being both a class and an object.

It can be seen that a *powertype pattern* is composed of the powertype itself (TreeSpecies in Figure 1), a type partitioned by it (Tree), and a relationship between them (see the Appendix for a completed definition of the Powertype pattern). This relationship can also be understood in terms of sets (Figure 2). The set of trees (the Tree class) comprises all trees (instances of the Tree class, represented as dots inside the ellipse on the left) and can be partitioned into subsets (subclasses of Tree) such as SugarMaple, Oak and Elm. Now each of these is a tree species such that we can construct a new set, the elements of which are all individual tree species —the TreeSpecies class on the right hand side of Figure 2, which contains the three elements of Elm, SugarMaple and Oak.
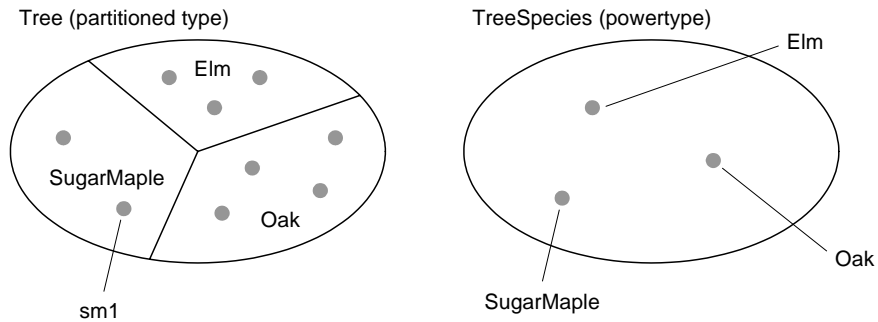
Figure 2. Two representations of trees: a tree species may be a set of instances within the Tree class i.e. a subset of Tree (left hand side) or may be a single instance in the TreeSpecies class (right hand side), which is called a powertype.

Thus, for example, Oak is an instance of the TreeSpecies class (right hand side) and also represents a subset of instances of the Tree class (left hand side of Figure 2). This duality is reflected in the different notation used in these two representational diagrams: Oak is represented as a "wedge" in the left hand diagram of Figure 2 and as a dot in the right hand diagram. A second example in a more technical IT domain, that of process modelling, is given in Figure 3.
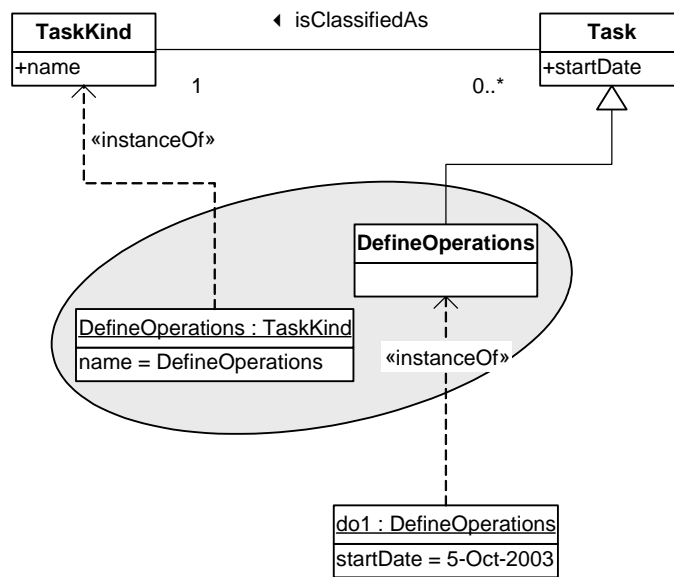

(a)
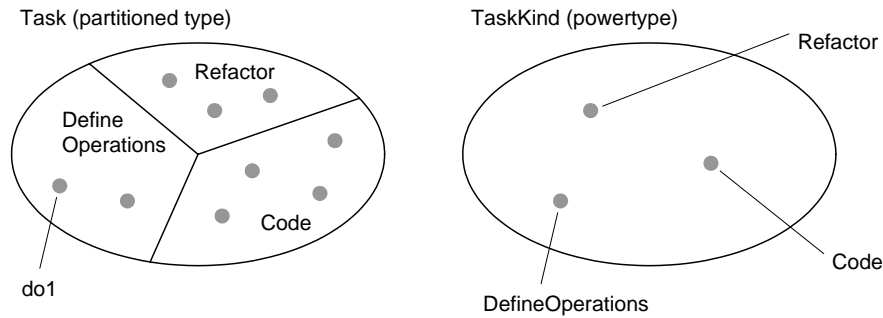
(b)

Figure 3. A second example, this time in the process modelling domain, illustrating (a) the powertype pattern between the various classes in the pattern and (b) the set-based equivalent.

These examples show the power of powertypes since not all subtypes need to be predefined as a class in a class model; in other words, powertyping enables *dynamic specialization*. A model is usually created by some authors and then used by a different set of persons. Traditionally, model authors depict specialization semantics using a conventional generalization relationship (such as the one between Boat and Vehicle in Figure 4). This can be called *static specialization* since, once modelled by the model authors, model users are given a set of classes that can be instantiated but not changed. Some examples are a class library in a programming environment (in which class and subclasses have been written and compiled by the class library author and are instantiable but not modifiable by the programmer) or a metamodel in a process engineering situation (in which a process engineer has created a predefined set of process elements that can be later instantiated by a project team). Using dynamic specialization, in contrast, the model author would represent specialization semantics using a powertype pattern, thus providing model users with a powertype of the generic concept (the partitioned type) from which subtypes of it can be instantiated at runtime. We must notice here that the terms "design time" and "runtime", usually applied only in the context of programming, are also useful to describe the design and building of a model (the model's "design time") and the subsequent usage of the model (the model's "runtime").

The powertype pattern can, in fact, be applied to just about every regular subtyping hierarchy. For example, a common example is that in the motor vehicle domain. Figure 4 shows the normal model, using inheritance (UML's generalization relationship), for a Vehicle with a subtype of Boat, an instance of which is also shown in this figure using regular UML notation. However, it is perhaps more accurate to introduce the class VehicleKind (of which Boat is just one instance), where VehicleKind classifies/categorizes Vehicle (Figure 5). Boat can then be instantiated to create objects such as b2:Boat (Figure 5 shows b2 as both using the UML object notation (a) and as an element of a set (b)). Since instantiation occurs at runtime, we can dynamically create other instances of VehicleKind, such as Bicycle or Car, in parallel to Boat in Figure 5(a). By induction, we therefore propose that any (or, rather, most) normal inheritance hierarchies in OO models can be easily modified to become powertype patterns, which
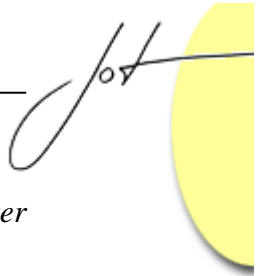
thus represent features that belong to the powertype (such as the *canTravelOnWater* attribute) as well as those of the partitioned type.
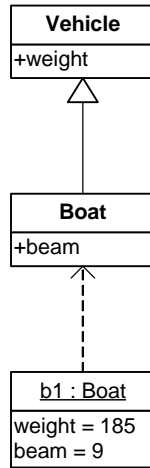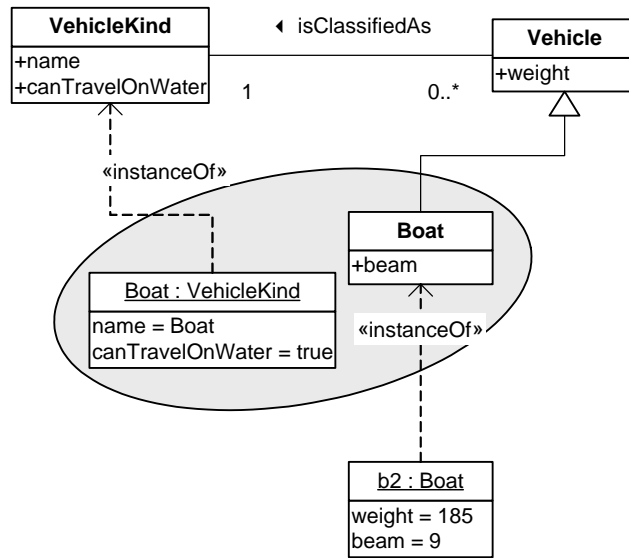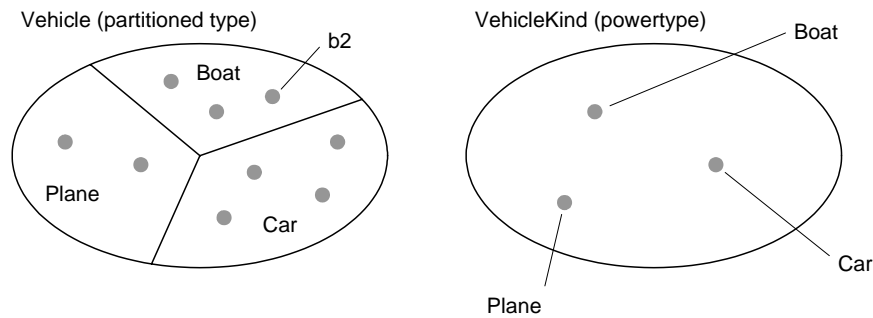

Figure 4. Regular UML model, using generalization, for Boats and Vehicles.

(a)

(b)

Figure 5. Re-modelling the example in Figure 4 using (a) a powertype pattern and (b) a set-based representation.

We have been using an ellipse to denote a concept that is represented in a model by a class and an object at the same time. Such hybrid class/object modelling entities have been named "clabject" by Atkinson and Kühne (2000) and we will use this concept and terminology in the remainder of this paper. Consequently, our notational ellipse, as used in diagrams to show a concept that is simultaneously modeled as class and object, actually depicts a clabject.

3   STEREOTYPES

Stereotypes comprise an extension and customization mechanism introduced, defined and used by the UML. Figure 6 shows the standard UML definition of a stereotype in which the value of the so-called "baseclass" (here the metaclass "Class") constrains the type of M1 elements to which this particular stereotype can be applied (in this example only to classes).
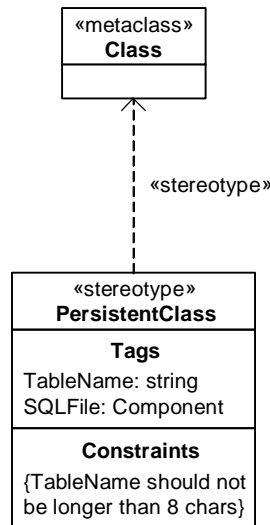
Figure 6. Definition of a stereotype in UML.

In contrast to powertypes, they are not a generic modelling mechanism but a UML-specific way of extending the UML metamodel. A *stereotype*, as defined in the UML 1.4[1] (OMG, 2001), is a virtual, user-defined metasubtype[2]. This is a subtype of some metamodel class that the user would ideally like to add to the UML metamodel. However, changing the UML metamodel by the UML user is not encouraged so, instead, he/she "pretends" to add it. In Figure 7, Class is a standard UML metamodel (M2 in OMG parlance) class. An instance of this might be the M1 level class called Bird. However, the user wishes to constrain the Bird class (and its instances) to be of a special kind of Class, in this example, an Entity Class. Thus, we invent the stereotype *entity* and define it to be effectively a subtype of Class called EntityClass. Since an M1 element is defined in UML to be an instance of an M2 element, the Bird class is both a class (from the modeller's perspective) and an instance of the appropriate M2 element (from the metamodeller's perspective), in our case, EntityClass (Figure 7). These two entities, however, are one and the same thing, so they are shown inside a greyed ellipse as before.

---

[1] Since UML 2.0 is still a draft document at the time of this writing and probably will not be finalized until the 3rd quarter of 2004, we consider it premature to incorporate it into our analysis in this current paper.
[2] It should be noted, however, that some modellers interpret the UML stereotype mechanism in a somewhat different fashion (Atkinson *et al.*, 2002).
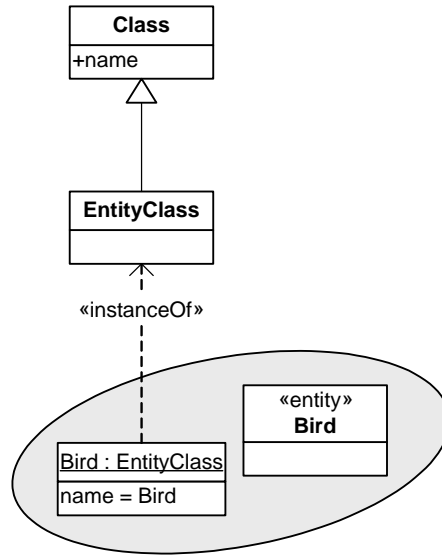
Figure 7. Adding a stereotype (virtual submetatype) to the UML metamodel.

A *stereotype pattern* is thus composed of an M2 class and a virtual metasubtype or stereotype (see the Appendix for a complete definition of the Stereotype pattern). If we now express this stereotype pattern in terms of sets (Figure 8), we see that the class Class and its three typical stereotypes of ControlClass, BoundaryClass and EntityClass look identical to the three divisions of Vehicle (Car, Boat and Plane) in Figure 5b. This leads us to construct the power set as seen in Figure 8. This set is clearly representative of a new class called ClassKind with its three members of ControlClass, BoundaryClass and EntityClass. Clearly, by analogy with Figure 5b, ClassKind must be a powertype i.e. a set of all subsets of another set as defined by a given discriminator.
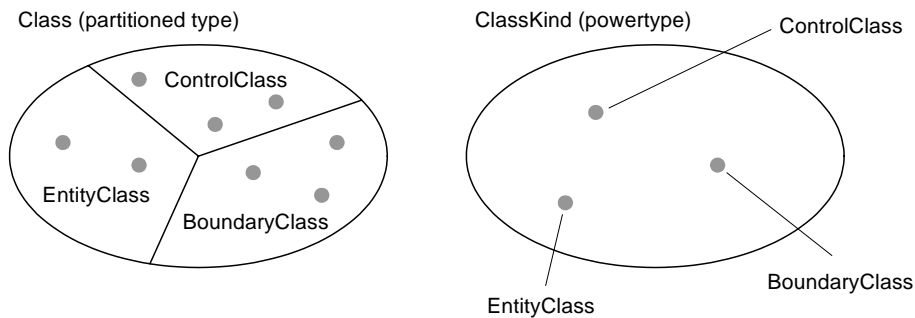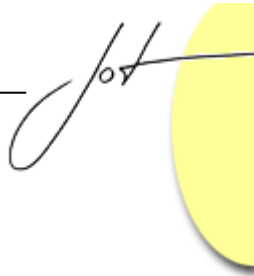

Figure 8. The partition into EntityClass, ControlClass and BoundaryClass expressed using a set-based notation.

4   LINKING POWERTYPES AND STEREOTYPES

Comparison of the set diagrams, particularly Figure 8 and Figure 5b, suggests that we should be able to find a mapping between these two patterns. This is accomplished by taking the stereotype pattern of Figure 7 and adding a single class, thus converting it into a powertype pattern. This is demonstrated in Figure 9, in which the class ClassKind (as represented by the set in Figure 8) augments the original stereotype pattern, thus creating the powertype pattern as seen earlier in Figure 5a.
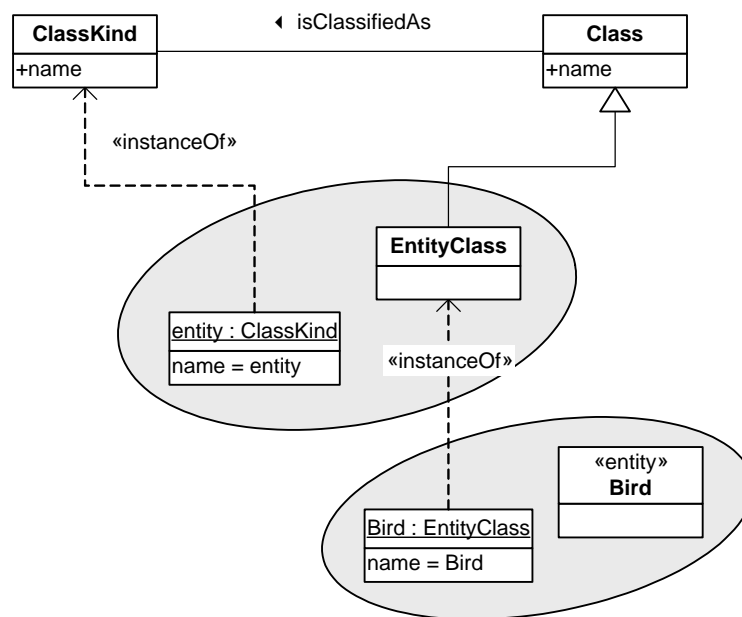

Figure 9. Taking the stereotype of Figure 7 and turning it into a powertype.

In general, a stereotype pattern can be expressed as a powertype pattern by explicitly representing the powertype class of which the stereotype is an instance (ClassKind in our example in Figure 9). This does not mean adding an extra element but explicitly showing an already existing element that is often taken for granted and not shown. Indeed, we conjecture, by inductive reasoning, that all stereotype patterns can be seen as powertype patterns. Of course, this "equivalence" is based on structural and not semantic arguments. Indeed, this is indicative of the need to complement this structural study with extensive semantic evaluations. At the same time, this structural equivalence should be taken into account when implementing metamodelling infrastructures in tools that store and manipulate models that use the constructs of powertypes and stereotypes. This will likely be of special importance for MDA-compliant tools[3].

---

[3] Model-Driven Architecture (MDA) is an initiative of the Object Management Group. Detailed information is available on http://www.omg.org
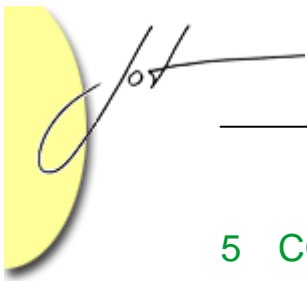
5   CONCLUSIONS

Powertype and stereotype patterns are very closely linked. Powertype patterns explicitly show one class more than the stereotype patterns. Once this class is explicitly shown in stereotype patterns, it is readily seen that the two patterns *do* become identical structurally. This similarity may be particularly useful in the context of tools that store and transform models containing both powertypes and stereotypes.

ACKNOWLEDGEMENTS

APPENDIX: PATTERN DEFINITIONS

This appendix offers definitions for the powertype and stereotype patterns based on the style proposed by Gamma *et al.* (1995).

Powertype Pattern

Intent. Support dynamic specialization by allowing subtypes of a class (the partitioned type) to be defined as instances of another class (the powertype) with regard to a given discriminant.

Motivation. Partitioning a class into subclasses is conventionally achieved through subtyping. With this approach, each subtype is implemented as a class, which (a) must be defined at design time and (b) cannot carry value slots. Defining subtypes as clabjects, the advantages of subtypes as classes remain and are combined with the advantages of subtypes as objects, namely the ability to be created at run-time and to carry slot values.

Applicability. Situations in which the subtypes of a certain type cannot be known or specified in advance.

Structure.

Powertype	◄ isClassifiedAs	PartitionedType
+discriminant		
	1 0..*	
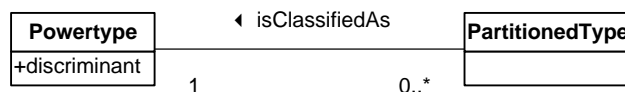
Participants and Collaborations. The PartitionedType class represents the class being dynamically subtyped. The Powertype class represents the subtypes that are dynamically created. Each instance of PartitionedType is classified as being of a specific subtype, i.e. a specific instance of Powertype, with regard to a given discriminant.

Example.
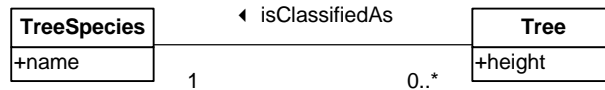
In this example, TreeSpecies is the powertype, whose instances represent specific species of tree (such as Sugar Maple, Elm or Oak). Tree is the partitioned type, since its instances are partitioned as dictated by the instances of TreeSpecies with regard to the discriminant name.

Consequences. By using this pattern, subtyping is no longer limited to design time. Subtypes of classes can be introduced dynamically as necessary. Each subtype thus introduced is implemented as a clabject, incorporating an object facet (instance of the powertype) and a class facet (subtype of the partitioned type).

Known Uses. This pattern has shown to be useful in metamodelling software development methodologies (Gonzalez-Perez and Henderson-Sellers 2005; SA 2004), since it can model both the methodology and project layers at the same time.
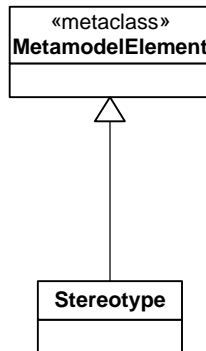
This pattern may be also used in advanced modelling situations in which subtypes of a given type cannot be hard coded at design time and therefore must be dynamically created at run-time.
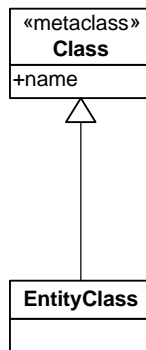
Stereotype Pattern

Intent. Customize an element in the UML metamodel for a particular usage.

Motivation. The elements in the UML 1.4 metamodel are generic enough to be applicable to a large number of situations. However, sometimes it is necessary to add details to a metamodel element so that it is optimized for a particular range of situations. Since the UML metamodel itself must not be changed, using stereotypes allows metamodel users to attain similar outcomes using only a "virtual" change, achieved through specialization.

Applicability. Situations in which a UML metamodel element is too generic and needs detail to be added for a particular usage.

Structure.

```
        «metaclass»
      MetamodelElement
      ──────────────────
              △
              │
              │
          ┌────────────┐
          │ Stereotype │
          ├────────────┤
          │            │
          └────────────┘
```

Participants and Collaborations. The MetamodelElement class represents the UML metamodel class that needs to be customized for a particular usage. The Stereotype class represents the optimized class.

Example.

```
        «metaclass»
          Class
        ──────────
          +name
            △
            │
            │
        ┌────────────┐
        │ EntityClass│
        ├────────────┤
        │            │
        └────────────┘
```

In this example, Class is the metamodel element being customized, and EntityClass is the customized version, representing a particular kind of class.

Consequences. By using this pattern, users of the UML metamodel (software developers and modellers) can add detail to any metamodel element for their own purposes, without changing the metamodel whatsoever.

Known Uses. This pattern is extensively used by UML itself in the definition of a number of metamodel elements such as instance-of dependencies, entity classes or library components.

This pattern is also used by modellers to optimize UML metaclasses for specific purposes.
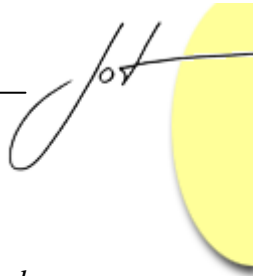
REFERENCES

Atkinson, C. and T. Kühne, 2000. Meta-Level Independent Modelling. In *International Workshop on Model Engineering at 14th European Conference on Object-Oriented Programming*. Sophia Antipolis and Cannes, France, 12-16 June 2000.

Atkinson, C., T. Kühne, and B. Henderson-Sellers, 2002, Stereotypical encounters of the third kind, UML 2002 - The Unified Modeling Language (eds. J.-M. Jezequel, H. Hussman and S. Cook), LNCS Volume 2460, Springer-Verlag, Berlin, 100-114

Gamma, E., R. Helm, R. Johnson and J. Vlissides, 1995. *Design Patterns.* Addison-Wesley.

Gonzalez-Perez, C.A. and B. Henderson-Sellers, 2005. A Powertype-Based Metamodelling Framework. *Software and Systems Modelling*. [in press]

Odell, J. J. 1994. Power types. *Journal of Object-Oriented Programming*, 7(2), 8-12.

OMG, 1999. OMG Unified Modeling Language Specification, Version 1.3, June 1999, OMG document ad/99-06-09 [released to the general public as OMG document formal/00-03-01 in March 2000]. [Online]. Available http://www.omg.org

OMG, 2001. OMG Unified Modeling Language Specification, Version 1.4, September 2001, OMG document formal/01-09-68 through 80 (13 documents) [Online]. Available http://www.omg.org

OMG, 2002. Meta Object Facility (MOF) Specification. OMG document formal/2002-04-03 [Online]. Available http://www.omg.org

SA, 2004. Standard Metamodel for Software Development Methodologies. SA document AS 4651-2004.

About the authors
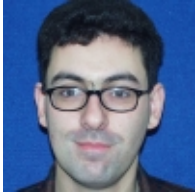
Cesar Gonzalez-Perez is a Post-doctoral Research Fellow at the Centre for Object Technology Applications and Research at University of Technology, Sydney (UTS), and has been developing and applying OO methodologies for over ten years to both research and commercial projects. He is the lead author of the OPEN/Metis methodology. E-Mail: cesargon@it.uts.edu.au
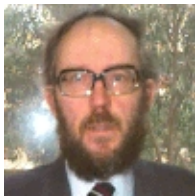
Brian Henderson-Sellers is Director of the Centre for Object Technology Applications and Research and Professor of Information Systems at University of Technology, Sydney (UTS). He is author of ten books on object technology and is well known for his work in OO methodologies (MOSES, COMMA and OPEN) and in OO metrics. He was recently awarded a DSc degree by the University of London for his work in object-oriented methodology. E-Mail: brian@it.uts.edu.au