On the Ease of Extending a Powertype-Based Methodology Metamodel

Brian Henderson-Sellers, Cesar Gonzalez-Perez

Faculty of Information Technology
University of Technology, Sydney
PO Box 123, Broadway, NSW 2007, Australia
brian@it.uts.edu.au
cesargon@verdewek.com

Abstract: Metamodelling is an increasingly prevalent tool in conceptual modelling – in particular in the context of OMG standards such as UML, MOF and SPEM. However, when applying a standard metamodelling approach based solely on instantiation semantics, many problems arise. These are shown to be solved using a powertype-based approach instead. Here we summarize this approach and focus on the ease with which this meta-architecture can be extended to support additional attributes and subtypes. This extensibility is readily accommodated within the strictures of the new metamodel without the need to invoke extension mechanisms such as stereotypes and profiles (as is currently advocated in the UML and SPEM).

1 Introduction

Metamodelling has been an instrument used increasingly over the last decade. Following suggestions [He94] that it might be useful as an underpinning for creating a coalescence of the then disparate set of OO modelling notations, it has now become enshrined in many OMG standards. However, when applied to the process component in methodology modelling, some problems emerge [AK01b]. The most important of these from a practical viewpoint is the effect of the so-called strict metamodelling hierarchy [At97], [AK00b] on the transmission of data values across the multiple layers in the now-traditional four-metalevel hierarchy, as used in OMG's UML, MOF and SPEM.

Put simply, any attribute declared at level M_n *must* have a value allocated to it at level M_{n-1} following the instantiation rules in use in strict metamodelling that dictate that only instance-of relationships can exist between adjacent metalevels. While this is fine for a modelling language, which is essentially defined at OMG level M_2 and used at level M_1, for process standardization the definitions at level M_2 are only realized (enacted) on real projects at level M_0 i.e. one level lower than the M_1 enactment layer for modelling e.g. [AK01a]. Thus, variables defined at level M_2 have values at M_1 and variables defined at M_1 have values at M_0. This is out of step with the notion that a standards body, like the OMG, wishes to standardize and "freeze" concepts at the M_2 level whereas the development team wish to use those concepts with real values at the M_0 level.

One way around this dilemma was proposed in [AK01a] in their attribution of a "potency" value to any meta-attribute that needed to have its value allocated more than the usual one level below its definition. This essentially allows a mixture of (implicit) generalization and instantiation semantics [GH05a], seemingly in defiance of the rules of strict metamodelling (although we note that Atkinson and Kühne [AK00a] go so far as to observe that the OMG's UML, while espousing a use of strict metamodelling, seems rather to use it very loosely, with instances and their types mixed together in the same level e.g. class and object, association and link).

Rather than starting with an axiomatic assumption of strict metamodelling and then trying to find ways to support double layer instantiation within that strict framework, as exemplified by Atkinson and Kühne's [AK01a], [AK01b] work, Gonzalez-Perez and Henderson-Sellers [GH05a] have proposed the use of powertypes [Od94].

In this paper, we summarize their proposals for using powertypes in full methodology (product plus process) modelling (Section 2 and 3). We then investigate, for the first time, the claim that such a powertype-based metamodelling framework is more flexible, specifically in terms of adding meta-attributes (Section 4.2) and extending the metamodel (Section 5). Such extensions in the UML world have been done by the increasingly complicated and poorly understood facility of stereotypes [AKH03]. Here we demonstrate how the adoption of a powertype-based metamodelling framework, instead of the strict metamodelling hierarchy, provides the much-needed and simple flexibility of extensibility sought after by OMG modellers.

2 Powertype Patterns

A powertype is defined as a type whose instances are subtypes of a second type called a partitioned type. The relationship between these two types, from an intuitive point of view, can be described as of a classificatory nature, although it can be argued that it carries instantiation semantics to a certain degree. However, it is not the aim of this paper to describe this relationship in depth but to use it. A powertype, together with the type that it partitions, plus the classificatory relationship between them, is called a powertype pattern (Figure 1).

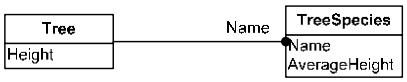


Figure 1. A powertype pattern, including a powertype (TreeSpecies) and a partitioned type (Tree). The powertype class is indicated by a black dot on its end of the line depicting their relationship.

Powertype patterns are slightly more complex than this. The partitioning that the powertype exerts over the partitioned type is not random, but obeys a well defined criterion. This means that instances of the partitioned type can be organised in groups (partitions) according to the value of a given attribute (often called a discriminant) of the powertype [HE94], [KM93], [OMG99]. In our example, TreeSpecies.Name is the discriminant of the powertype pattern formed by Tree and TreeSpecies (Figure 1); this

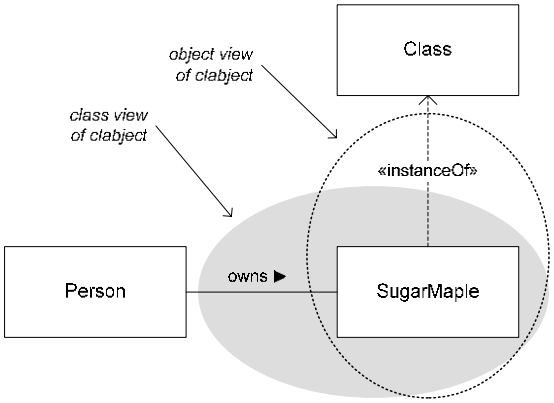
means that each instance of TreeSpecies (which will have a particular and unique value for the Name attribute) represents a collection of instances of Tree, namely, those trees of the given species.

The special nature of the classificatory relationship between the types involved in a powertype pattern raises an interesting issue: instances of the powertype and subtypes of the partitioned type represent the same thing. Following our example, instances of TreeSpecies represent specific species of tree, such as Oak or Sugar Maple. At the same time, subtypes of Tree (that use the powertype pattern's discriminator) also represent specific species of tree. For example, the concept of Sugar Maple can be represented by an instance of TreeSpecies but also by a subtype of Tree (Figure 2).

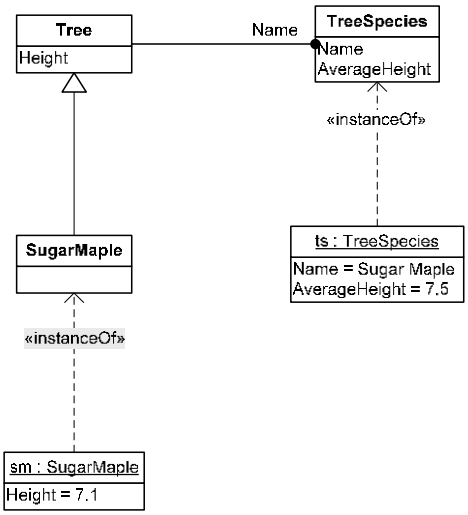
In Figure 2(b) using traditional (OMG) terminology, sm: SugarMaple would be at level M_0; SugarMaple at M_1 and TreeSpecies and Tree at M_2. However, this does not agree with the basic rule of strict metamodelling i.e. that generalization relationships are only permitted within a metalevel and not across metalevels. Furthermore, in the OO philosophy, an instance of a class is always an object whereas a subtype of a class is always a class. Thus, from the right hand side of Figure 2 we would deduce that SugarMaple is a class because of its generalization relationship to the class Tree but, contrariwise, from the left hand side, that SugarMaple is an object because it is an instance of a class (the class TreeSpecies). So, is SugarMaple a class or an object? Actually it is neither; SugarMaple is a concept. Whether we choose to model it as a class or as an object is just a matter of convenience. Using powertype patterns it is clear how we can use both representations at the same time, so, for the purpose of this paper, we can say that SugarMaple "is" both a class and an object. Here we use the term introduced in [At98] of "clabject" to illustrate a conceptual entity that has both a class and an object facet, like SugarMaple in our example. It should furthermore be noted that this class/object dilemma abounds in the OO literature but is generally ignored [GH05b].

Since UML, while mentioning powertypes, only uses a stereotype notation for the powertype itself, it does not offer a notation for the classificatory relationship within the powertype pattern nor for an entity that is both class and object, here, for illustrative purposes only, we will represent clabjects as an individual object and an individual class inside an ellipse (Figure 3).

It is possible to look at the same issue from a set theoretical perspective. The set of trees (roughly mappable to the Tree class) comprises all trees (instances of the Tree class, represented as dots inside the ellipse on the left in Figure 4) 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 set on the right hand side of Figure 4, which



(a)



(b)

Figure 2. (a) A powertype pattern allows the representation of the same thing as both a class and an object. (b) Using more standard notation, the concept of Sugar Maple is modelled in the figure as an instance of TreeSpecies and also as a subtype of Tree. A specific sugar maple, sm, is also shown.

contains the three elements of Elm, SugarMaple and Oak. Thus, for example, Oak is an element in the TreeSpecies set (right hand side) and also a subset of the Tree set (left hand side). 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 4 and as a dot in the right hand diagram. A second example in a more technical software engineering domain, that of process modelling, is given in Figure 5.

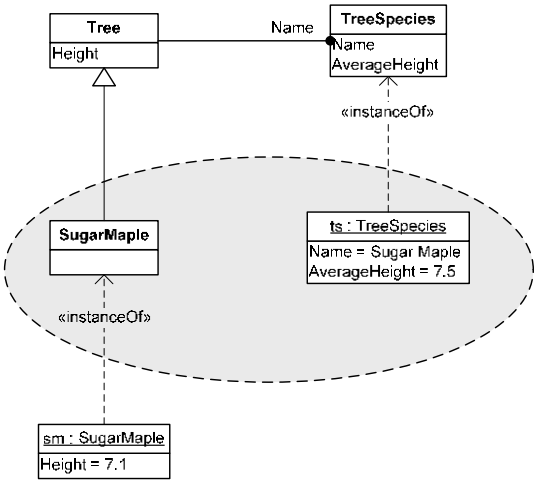


Figure 3. SugarMaple is both a class and an object, so a clabject is used to represent it. An ellipse is used to depict the clabject since UML does not offer a notation for a model element being both a class and an object (after [GH05c]).

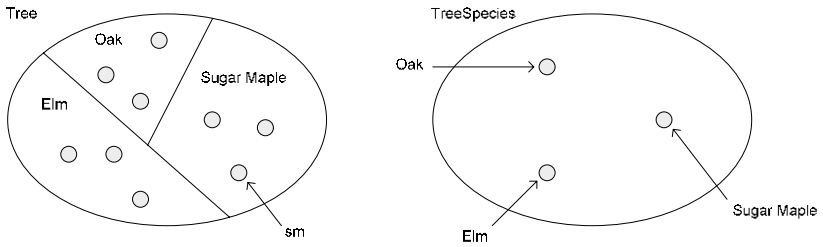
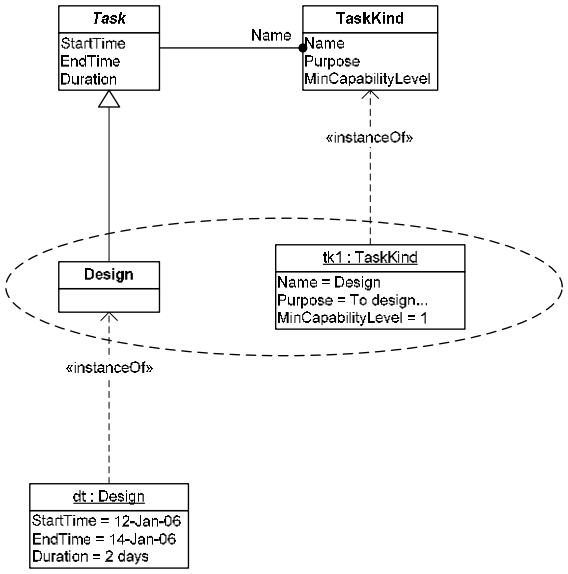


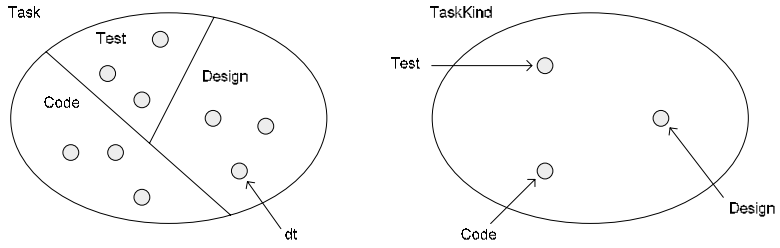
Figure 4. Two representations of tree species: a tree species may be represented as a subset of the Tree set (left hand side) or as a single element in the TreeSpecies set (right hand side).

Figure 5 shows with clarity how two different sets are possible: one in which there are a large number of elements (corresponding to the Tree or Task class in our examples) and a second which contains elements which are themselves of a classification nature (TreeSpecies or TaskKind). As we have seen, these latter are the powertypes. However, what is important in software engineering and modelling is that, while most people would readily discriminate between a tree and a tree species conceptually, the same cannot be said for software developers and modellers - with the exception of a small team who identified a similar notion in data modelling that they called materialization [DP02], [Pi94]. Our survey of the literature finds a large number of examples in which the two sets (Task and TaskKind in Figure 5) are convoluted. The same name (Task) is used for both, which means that in some arguments classes such as Task become a chameleon. Sometimes, Task is used to refer to the tasks that are actually performed by developers (with a duration and a start and end dates), and at other times the word Task is used incorrectly for the powertype class, which represents *kinds* of tasks and not the tasks themselves, very much like a TreeSpecies does not represent actual trees but

species of trees. Dissociating these two, now very obviously distinct, meanings (and their corresponding class notation within the OO context) opens the way for their use in methodology metamodelling.



(a)



(b)

Figure 5. A second example, this time in the process modelling domain, illustrating (a) a powertype pattern between Task and TaskKind and (b) the set-theoretical equivalent.

3 Applying Powertypes to the Methodology Domain

We have noted that for constructing a methodology and using it on real projects, three metalevels are required and utilized. However, any individual tends to focus on only two of these for any specific purpose. The methodologist or method engineer is concerned with creating a methodology for the organization or project but is usually unconcerned

about the enactment of it; whereas software developers and project managers have little use or cognizance of the highest (metamodel definitional) layer [GH09a]. Thus, rather than defining a three layer hierarchy in terms only of "instance-of" relationships (as done in strict metamodelling and the OMG framework), the three layers are delineated in terms of this industry practice viz. project-level (called here endeavour), methodology level (called here simply method) and metamodel layer (Figure 6). The link between each pair of layers can be described at this stage as a representation, without further details of its nature [HG05b] – see also [Se03].

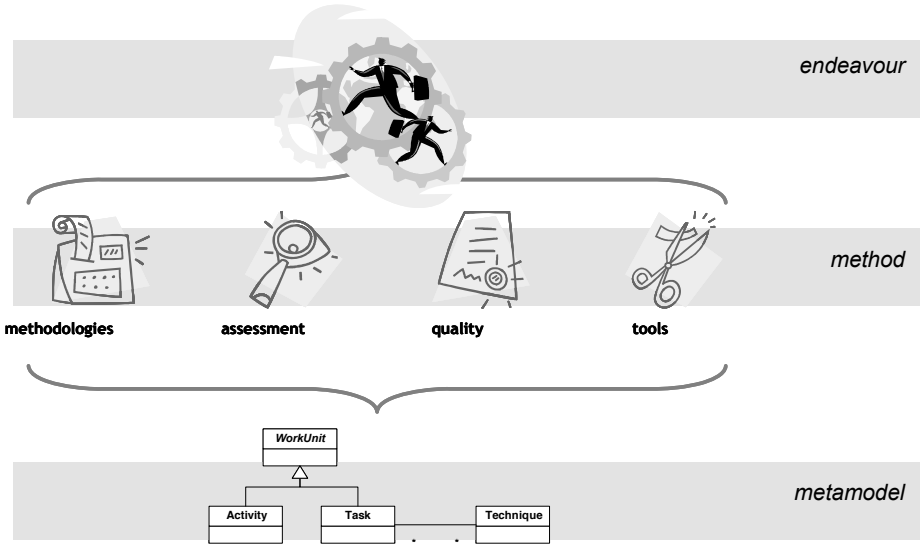


Figure 6. The new three layer architecture to support used in conjunction with powertype patterns.

This new layering architecture, more closely aligned with reality and practice (rather than with accidental, non-essential restrictions of specific modelling techniques), can now be shown to be consistent with the notion of a powertype pattern, as explained in Section 2. Concepts relating to the methodology specification are in the middle layer. Figure 7 shows one specific example, in which a Requirements Specification Document is included in the methodology as one of the possible kinds of work product. Such a method-level class typically has two different kinds of characteristics: (i) those relating to *all* such documents, which are a property of their class rather than the objects themselves and (ii) those that will always be present and therefore must be standardized yet not given values until the methodology is enacted on a particular project. This first kind of characteristic is modelled as an attribute of some class in the metamodel layer (here that class is called DocumentKind), which takes a value at the method level. The second kind of characteristic needs to be given a value at the project level yet standardized at the metamodel level, crossing over the method level untouched. As seen in Figure 7, this is accomplished by the use of a generalization relationship to a second metamodel class (here Document) and an instantiation relationship to the project level (here to the object with the title "MySystem" Requirements). It is now readily seen (by

comparison with Figure 2 *et seq.*) that these metamodel- and method-level classes form a powertype pattern together with some clabjects generated from it.

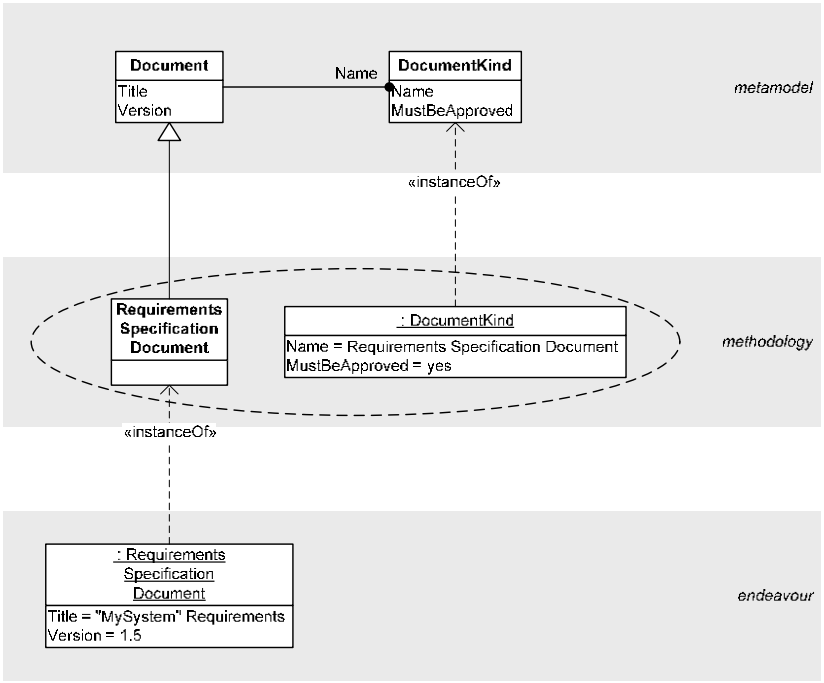


Figure 7. An example of a work product definition that uses a powertype pattern.

Finally, we might note that, although most of the classes in a metamodel will likely be involved in powertype patterns (either as powertypes or partitioned types), some classes will stand on their own. These classes can be instantiated by a method engineer into objects in the method layer, but they cannot be formally transmitted down to the endeavour layer. These classes, which we call *resources* [HG05a], represent elements in the methodology that are used by software developers without being instantiated (Figure 8), but only as a reference or guideline. Method-level entities that are instantiated into project elements (i.e. class facets of clabjects) are called *templates* [GH05c].

The use of powertypes, together with the templates/resources dichotomy, has been incorporated recently into the AS 4651-2004 standard [SA04] which itself became the base document for the ISO/IEC 24744 Project, "Software Engineering – Metamodel for Development Methodologies" (SEMDM in short), the core of which is depicted in Figure 9. The left hand side shows the top-level powertype patterns, while resource classes are shown on the right hand side. The scope of these standards encompasses both product and process as well as providing support for capability assessment.

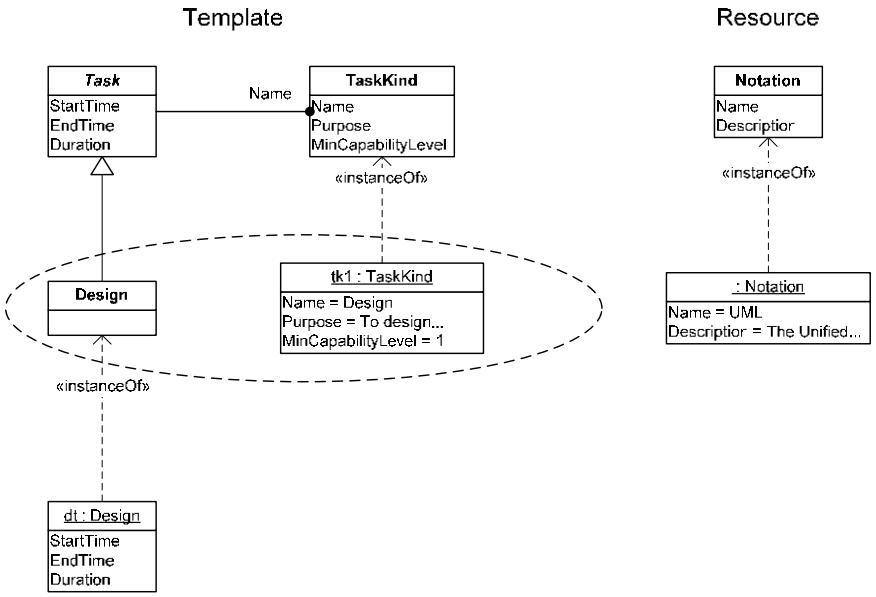


Figure 8. Examples of templates and resources (after [HG05b]).

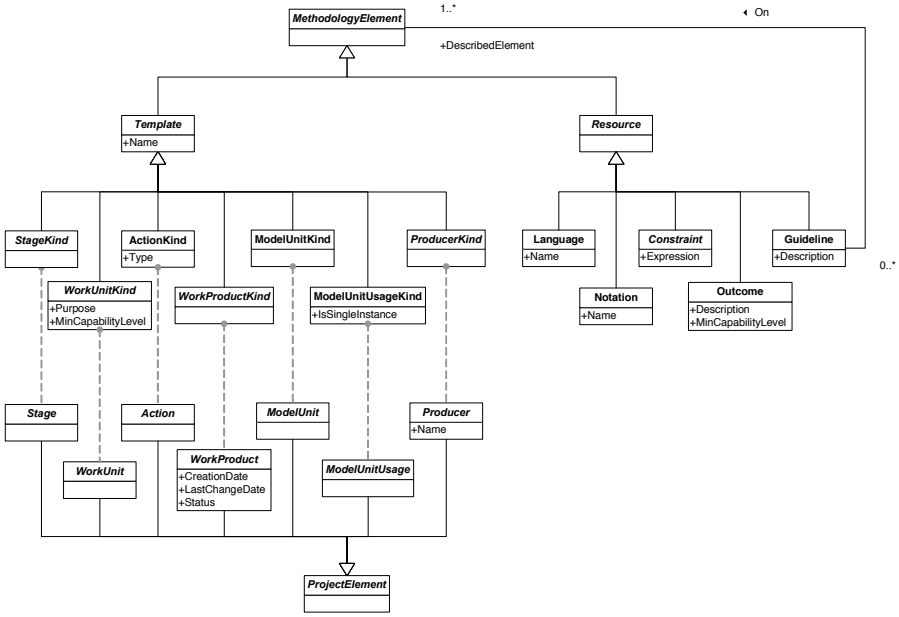


Figure 9. The core of the SEMDM.

4 Usage of a Powertype-Based Methodology Metamodel

The creation of a methodology from a metamodel requires the instantiation of each of the meta-elements into the elements that compose the methodology. With a method engineering approach, each metaclass can be considered independently, individual fragments instantiated and then the methodology constructed from these fragments by an appropriate configuration approach and/or tool.

With a conventional metamodel (i.e. one *not* employing powertypes), this process utilizes regular instantiation except that, strictly, the elements of the methodology instantiated from the metamodel are more classlike than objectlike [GH05b]. In contrast powertype instantiation combines regular instantiation semantics with generalization semantics thus permitting the creation of clabjects – these are the methodology elements we require for methodology construction.

In this paper, rather than expound upon the (powertype) instantiation of the whole methodology, we select, as an exemplar, a single meta-element, that of Task and its associated powertype class TaskKind. As noted above, the classes Task and TaskKind form a powertype pattern in the metamodel. TaskKind is the powertype of Task, and Task is called the partitioned type. TaskKind represents the kinds of tasks that can be defined at the methodology level, whereas Task represents the actual tasks that will occur at the endeavour level (i.e. when the methodology is enacted). The powertype pattern formed by Task and TaskKind can be denoted in shorthand as Task/*Kind (Figure 10).

4.1 Simple Usage

To use this for methodology construction, a method fragment is created by instantiating the powertype pattern. This is not a conventional OO instantiation but a more complex process. First of all, the partitioned type (Task in our case) is subtyped into a new class, ElicitRequirementsTask in our example. Then, the powertype is instantiated into an object, tk1 in our example. ElicitRequirementsTask and tk1 represent the same concept, i.e. tasks for requirement elicitation, but they represent it in different ways and for different purposes. Together, ElicitRequirementsTask and tk1form a clabject, indicated by an ellipse in the diagram. Notice how the object facet of the clabject describes the particular task kind by carrying values (for the attributes of TaskKind), while the class facet serves as a template for endeavour-level instantiation (see below).

A method fragment is used by developers by instantiating its class facet, ElicitRequirementsTask in our example. Notice how attributes for any task are defined by the metamodel in class Task and take values at the endeavour level, in the etk1 object in the example in Figure 10.

By using powertype patterns and clabjects, two sets of values exist: one, at the methodology level, describes the method fragment, and another, at the endeavour level, that describes the instances of this method fragment.

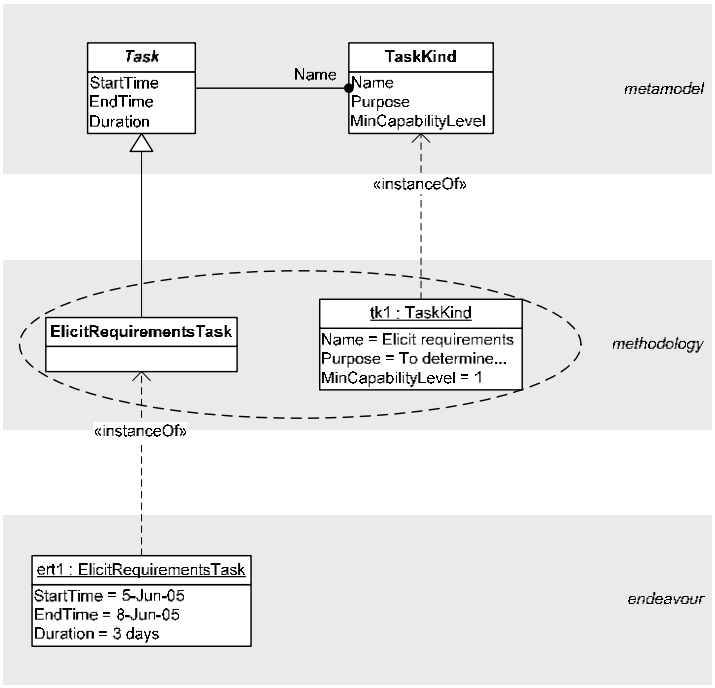


Figure 10. A powertype pattern in the task domain.

4.2 Adding an Attribute at the Method Level

Using a powertype-based metamodel, the methodologist has the opportunity to add attributes and associations to the classes in the methodology level as she/he defines them. This is not possible in a conventional approach because all attributes are frozen at the metamodel level. In this scenario, very similar to the previous one, a ValidateRequirementsTask method fragment is defined, and a NeedsSignOff attribute is to be added to it (Figure 11). This attribute represents that some requirement validation tasks, when actually performed at the endeavour level, will need signing off before the requirements can be considered validated. The methodologist captures this need of the methodology by incorporating this attribute into the class facet of the method fragment. In programming language terms, this is known as "programming by difference" and comprises a key strategy in the OO paradigm.

Notice how this attribute, NeedsSignOff, is used, together with the attributes generic for every task, at the endeavour level in object vrt1.

5 Extending a Powertype-Based Methodology Metamodel

So far we have explained how a powertype-based metamodel can be used "as is" to create method fragments. This section considers those cases when the metamodel is not fully appropriate for the task at hand and needs to be extended before it is used. In a conventional (e.g. OMG) approach, the only mechanisms available to provide extensibility are (1) extending the metamodel directly – thus going outside the standard and making the use of standard-compliant tools difficult; (2) extending the metamodel indirectly using OMG's UML stereotypes. Unfortunately, the use of stereotypes is ill understood, poorly applied and, in UML version 2.0, is found to violate its underpinning theory [HG05c].

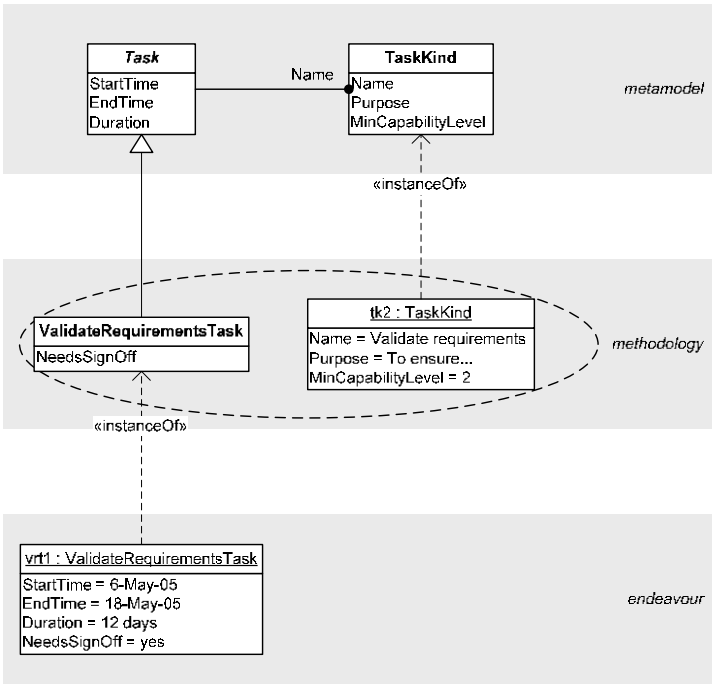


Figure 11. Adding a new attribute to a class at the method level.

In contrast, the powertype-based metamodel, as used in ISO/IEC 24744 (SEMDM), can be extended without transgressing its standardized nature, as follows. Consider that the methodologist decides that many tasks in projects that use the methodology he/she is designing will need to be measured for performance, and so decides to refine the standard Task/*Kind powertype pattern provided by the metamodel and create a MeasuredTask/*Kind subtype of it (Figure 12), which incorporates the necessary infrastructure to manage performance measurement of tasks. In order to subtype a powertype pattern, two new classes (called extension classes) are introduced, each being a subtype of the classes involved in the original powertype pattern (called standard classes). In this example, MeasuredTask is subtyped from Task, and MeasuredTaskKind

is subtyped from TaskKind. The powertype pattern relationship between the extension classes parallels the standard one in the metamodel.

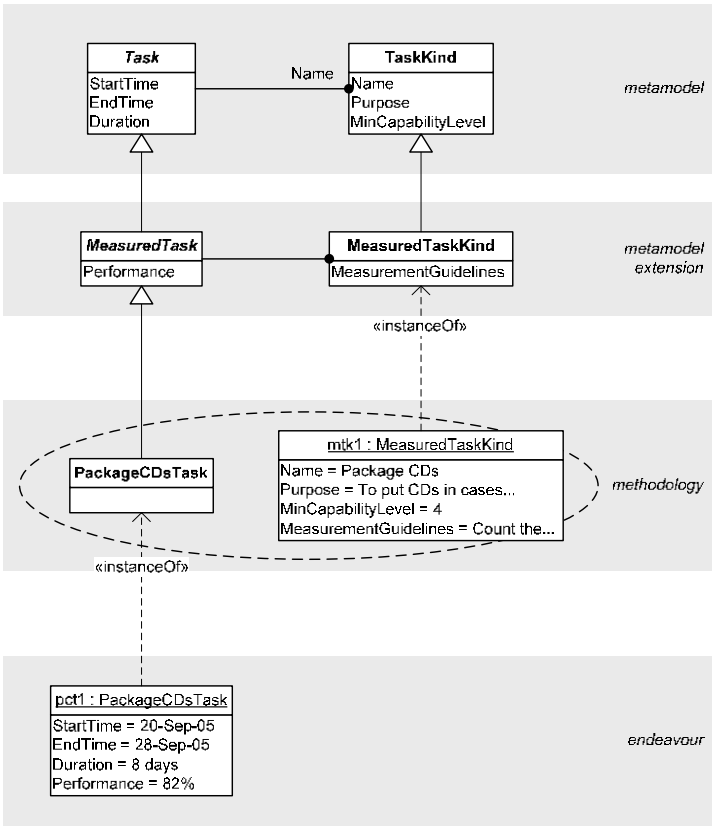


Figure 12. Metamodel extension followed by usage.

Note how MeasuredTask carries a Performance attribute, capturing the fact that, at the endeavour level, all measured tasks will have a performance value attached to them. Similarly, MeasuredTaskKind carries a MeasurementGuidelines attribute, which, for each particular kind of measured task, will contain information explaining how performance is to be measured.

Once the extension powertype pattern has been introduced, the methodologist (the same who introduced it or another one) can use it in the conventional way, deriving method fragments from it as explained in previous sections.

Notice how the attributes introduced at the metamodel extension level take values and are used at the methodology and endeavour levels.

The mechanism here described is in stark contrast to the invention of the notion of profiles and stereotypes in the UML. There, the metamodel cannot be extended without

changing the (standardized and essentially immutable) M_2-level metamodel. Here, an equivalent extensibility is accomplished without damaging the original metamodel yet permitting user-defined extensions (akin to a profile) all within the standard and supportable by tools. Furthermore, conventional, well-known OO building blocks such as attributes and specialization are used to extend the metamodel, rather than made-up constructs such as stereotypes and tagged values in the UML.

6 Conclusions

Metamodelling has become an increasingly accepted part of conceptual modelling, penetrating the standards of such organizations as the OMG and ISO. The approach used in OMG standards is that of strict metamodelling, which uses only instantiation semantics. While this works reasonably for UML and other modelling languages, it does not permit any degree of extensibility without invoking the notion of profiles and stereotypes (themselves widely debated and criticized and not part of conventional object-orientation); nor does it permit appropriate synergy between the product and process parts of a methodology metamodel, such as found in OMG's SPEM.

Here we have summarized a powertype-based metamodelling approach, discussed in detail earlier (e.g. [GH05a], [Od94]) before demonstrating and analysing the extension mechanisms appropriate to such an approach. In particular, we have shown how an attribute can readily be added to one of the metatypes in any standardized metamodel built in this way and, secondly, how the whole idea of profiles is obviated because the combined instantiation and generalization semantics that are a natural consequence to the use of powertype patterns allow the user to define subtypes within the "metamodel layer" because both type and powertype can be specialized within this layer and serve as an intermediate "extension layer" as demonstrated in Figure 12 and Section 5. As a proof of viability beyond the (mostly) theory presented here, powertype patterns are the basis of an ongoing project within the ISO community (ISO/IEC 24744).

References

[At97] Atkinson, C.: Metamodelling for distributed object environments, *Procs. First International Enterprise Distributed Object Computing Workshop (EDOC'97)*, Brisbane, Australia (1997)

[At98] Atkinson, C.: Supporting and applying the UML conceptual framework. In *«UML» 1998: Beyond the Notation*, Vol. 1618. Bézivin, J. and Muller, P.-A. (eds). Springer-Verlag, Berlin (1998) 21-36.

[AK00a] Atkinson, C., Kühne, T.: Meta-level independent modelling. In International Workshop on Model Engineering at 14th European Conference on Object-Oriented Programming (2000)

[AK00b] Atkinson, C., Kühne, T.: Strict profiles: why and how. In *«UML» 2000: Advancing the Standard*, Vol. 1939. Evans, A., Kent, S. and Selic, B. (eds). Springer-Verlag, Berlin (2000) 309-322.

[AK01a] Atkinson, C., Kühne, T.: The essence of multilevel metamodelling. In *«UML» 2001: Modeling Languages, Concepts and Tools*, Vol. 2185. Gogolla, M. and Kobryn, C. (eds). Springer-Verlag, Berlin (2001) 19-33

[AK01b] Atkinson, C., Kühne, T.: Processes and Products in a Multi-level Metamodeling Architecture. *Int. J. Software Eng. and Knowledge Eng.*, 11(6) (2001) 761-783.

[AKH03] Atkinson, C., Kühne, T.: Henderson-Sellers, B.: Systematic stereotype usage, *Software and System Modelling,* 2(3) (2003) 153-163

[DP02] Dahchour, M., Pirotte, A.: Materialization and its metaclass implementation. *IEEE Transactions on Knowledge and Data Engineering,* 14(5) (2002) 1078-1094.

[GH05a] Gonzalez-Perez, C., Henderson-Sellers, B.: A powertype-based metamodelling framework, *Software and Systems Modeling*, 4(4) (2005) DOI 10.1007/210270-005-0099-9

[GH05b] Gonzalez-Perez, C., Henderson-Sellers, B.: A representation-theoretical analysis of the OMG modelling suite, *New Trends in Software Methodologies, Tools and Techniques* (eds. H. Fujita and M. Mejri), IOS Press, Amsterdam (2005) 252-262

[GH05c] Gonzalez-Perez, C., Henderson-Sellers, B.: Templates and resources in software development methodologies, *J. Obj. Technol.*, 4(4) (2005) 173-190

[He94] Henderson-Sellers, B.: Methodologies - frameworks for OO success, *American Programmer,* 7(10) (1994) 2-11

[HE94] Henderson-Sellers, B., Edwards, J.M.: *BOOKTWO of Object-Oriented Knowledge: The Working Object*, Prentice-Hall, Sydney (1994)

[HG05a] Henderson-Sellers, B., Gonzalez-Perez, C.: Connecting powertypes and stereotypes, *J. Object Technol*., 4(7) (2005) 83-96

[HG05b] Henderson-Sellers, B., Gonzalez-Perez, C.: The rationale of powertype-based metamodelling to underpin software development methodologies, *Conferences in Research and Practice in Information Technology,* 43 (eds. S. Hartmann and M. Stumptner), Australian Computer Society (2005) 7-16

[HG05c] Henderson-Sellers, B. and Gonzalez-Perez, C., 2006, Uses and abuses of the stereotype mechanism in UML1.4 and 2.0, *Model Driven Engineering Languages and Systems, 9th International Conference, MoDELS 2006, Genoa, Italy, October 2006* (eds. O. Nierstrasz, J. Whittle, D. Harel and G. Reggio), LNCS, Springer-Verlag, Berlin.

[KM93] Korson, T.D., McGregor, J.D.: Supporting dimensions of classification in object-oriented design, J. Obj.-Oriented Progr., 5(9) (1993) 25-30

[Od94] Odell, J.J.: Power types. *Journal of Object-Oriented Programming,* 7(2) (1994) 8-12.

[OMG99] OMG: OMG Unified Modeling Language Specification, Version 1.3, June 1999 (1999)

[Pi94] Pirotte, A., Zim□nyi, E., Massart, D., Yakusheva, T.: Materialization: a powerful and ubiquitous abstraction pattern. In *20th International Conference on Very Large Data Bases.* Bocca, J., Jarke, M. and Zaniolo, C. (eds). (1994) 630-641.

[Se03] Seidewitz, E.: What models mean. *IEEE Software,* 20(5) (2003) 26-31.

[SA04] Standards Australia: Standard Metamodel for Software Development Methodologies, AS 4651-2004 (2004)