Appeared in the International Journal of Software Engineering and Knowledge Engineering, Vol. 11, No. 6, p. 761–783, 2001

Processes and Products in a Multi-Level Metamodeling Architecture

Colin Atkinson and Thomas Kühne

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

Abstract. Following the successful use of object-oriented metamodeling in the definition of the UML and other notation standards there is increasing interest in extending the approach to cover other concepts of software development, including processes. However, it turns out that the "obvious" approaches for using metamodels to describe processes and artifacts independently do not integrate well together in a natural and straightforward way. In this paper we discuss the problems and inconsistencies than can arise when trying to model a process and the products it creates within the same metamodeling framework, and present a solution that not only avoids many of these problems but also qualifies as a general metamodeling pattern. We then generalize the conceptual architecture to support the sound co-modeling of all independent areas of concern within the context of strict metamodeling.

Introduction

Object-oriented metamodeling has become an increasingly important tool in the definition and presentation of software development concepts. Following the successful use of metamodeling in the definition of the UML [1], there is a growing interest in using a similar approach to describe the process aspects of software methods as well as the notational aspects. However, this is not as straightforward as it might at first appear. Describing a process within a four-layer metamodeling architecture is not a problem as long as the products of the process are viewed as primitive artifacts. However, as soon as an attempt is made to link the metamodel of the process to the metamodel of the product subtle problems arise. These have to do with the relative location of modeling elements and the nature of the relationships between them. These problems can break the logical consistency of the metamodels and undermine their value as the backbone for semantics and tools support

The goal of this paper is to illuminate the nature of the problems that arise when processes and their products are defined within the same multi-level metamodeling architecture, and to describe how these problems can be overcome. We then show how the solution can be generalized into a metamodeling pattern. We conclude by presenting a conceptual view which allows a clean interpretation of modeling scenarios where the pattern is not applicable and which otherwise have no well-defined meaning.

Modeling Processes and Products

The typical four-layer metamodeling architecture used in the definition and application of modeling languages, such as the UML[1] and the OML[2], is depicted in **Fig. 1**. The bottom two levels correspond to the traditional class/instance levels familiar to object-oriented developers. The third level, M_2, is known as the Metamodel, and describes the modeling elements of which the typical user classes are instances. Finally, the top level, M_3, is known as the Meta-Metamodel and defines the concepts of which M_2 level artifacts are instances.

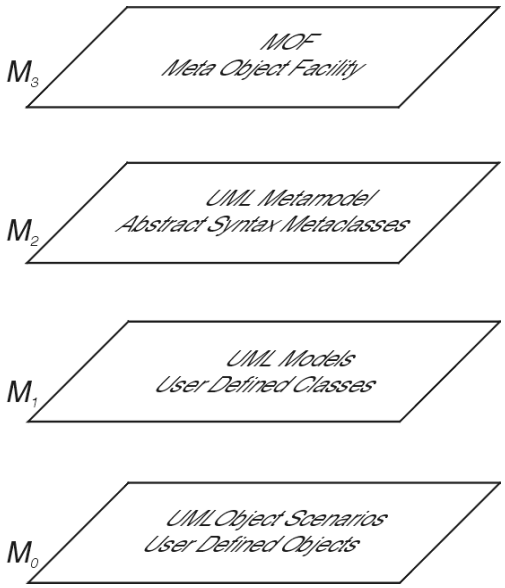


Fig. 1. Four-layer Metamodeling Architecture

Modeling Products

Fig. 2 illustrates the basic strategy used by the UML to relate artifacts, and their modeling concepts within the four-layer architecture. The basic idea is that objects at the M_0 level, such as `BobsUncle`, are instances of classes at the M_1 level, such as `Person`, which in turn are instances of M_2 level classes such as `Class` and so on. Also, the class diagram concept (`ClassDiagram`) is composed of multiple class concepts (`Class`), as indicated at the M_2 level, while user class diagrams, such as `BobsClassDiagram`, are made up of user classes, as indicated at the M_1 level.

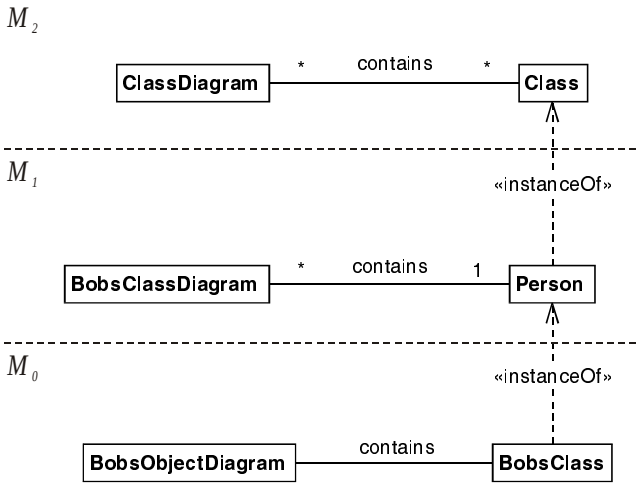


Fig. 2. Artifact Description within a Four-layer Architecture

Modeling Processes

A similar strategy can be used to describe a process within a multi-level modeling architecture, as illustrated in **Fig. 3**. Although artifacts (i.e., process products) are mentioned in this approach, no attempt is made to describe their content. In other words, they are regarded as primitives. Usually, as in this case, this is not a problem since there is no need to explicitly model the contents of the artifact.

As can be seen from this figure, specific executions of a process, or activity, such as `BobBakesCake`, appear at the M_0 level, while activity types (or templates) appear at the M_1 level. The generic concept of `Activity` is represented as a Metaclass at the M_2 level. This approach is fairly natural, and with one minor exception provides all the required expressive power. The exception relates to the lack of transitivity of attributes and relationships with respect to the instance of relationships [3]: Because attributes and relationships of a class become slots and links of its instances, they can have no impact on further instantiation. For example, although it seems to be desirable to locate attribute `duration` at the M_2 element `Activity` so that all activities automatically have this feature, this is not possible. Attribute `duration` would become a slot of `BakeCake` and as such would not be passed on to the `BobBakesCake` instance. The only way in the scenario of **Fig. 3** to create a duration slot in `BobBakesCake` is to define it as an attribute in `BakeCake`. We will come back to this issue later in the paper.

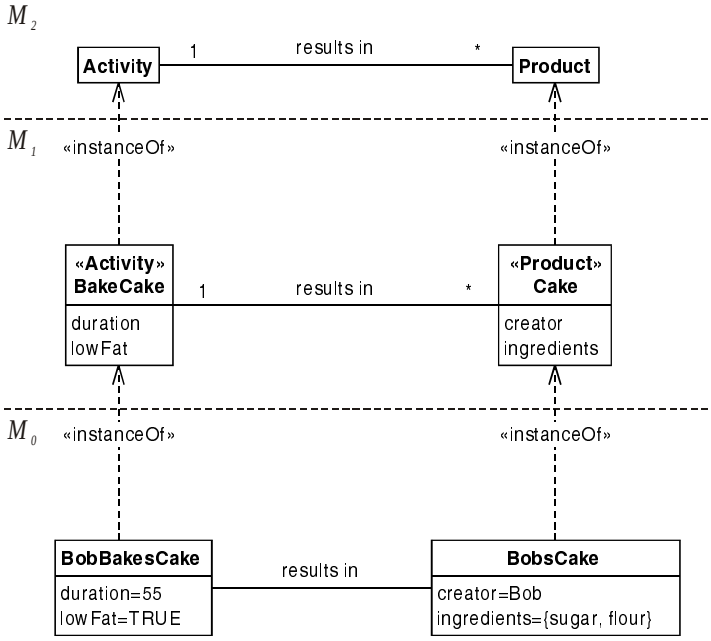


Fig. 3. Process Description within a Four-layer Architecture

Combining Processes and Products

A more serious problem arises if one attempts to describe *both* the contents of artifacts and the producers of artifacts within the same modeling framework. The essential difference to the process modeling we discussed before is that the products of activities are no longer viewed as primitive artifacts, but as complex artifacts whose contents are captured by relationships to other modeling elements. As illustrated in **Fig. 4**, the heart of the problem is that the natural level at which artifact contents are represented within the four-layer metamodeling architecture may be different depending on whether they are viewed as part of the process or the product meta hierarchy. The situation of specific user class diagrams is a prime example: From the process perspective, a specific class diagram, such as `BobsClassDiagram` (see **Fig. 4**), should reside at the M_0 level, since it is produced by a specific execution of a process activity, in this case `BobsDesign`. However, from the product content perspective, a class diagram, such as `BobsClassDiagram`, should reside at the M_1 level, since it is made up of M_1 level classes, such as `BobsClass`.

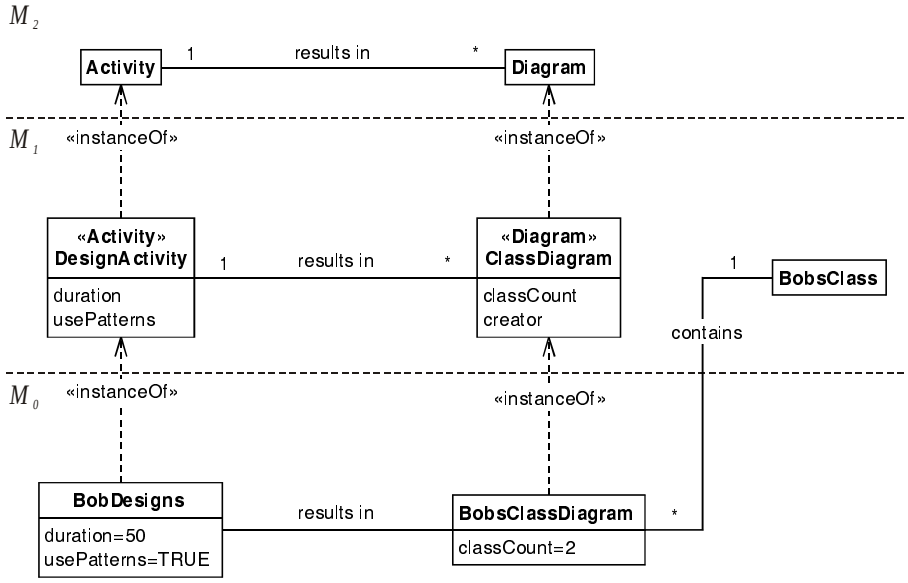


Fig. 4. Activity Products as Complex Artifacts

The fundamental question is how to deal with this situation. The answer depends on how serious one is about the multiple levels of the four-layer modeling architecture, and what properties one would like the final modeling architecture to exhibit. A straightforward answer would be to arbitrarily select a level for `BobsClassDiagram` and ignore the fact that it initially seemed to possess two "natural" levels. To see why this is not an entirely satisfactory solution and understand the range of available solution strategies it is necessary to appreciate the distinction between strict- and loose metamodeling. This is discussed in the following section.

Strict- versus Loose Metamodeling

Two different schools of thought exist with regard to the importance of level boundaries in a multi level metamodeling architecture: Strict metamodeling [4] holds that level boundaries are real and can only be crossed by instance-of relationships whereas loose metamodeling regards the levels merely as a helpful organisation principle but allows relationships to cross level boundaries in arbitrary ways.

Loose Metamodeling

Loose metamodeling essentially encompasses all approaches which claim that one model is an instance-of another model, but where the instance-of relationship between individual model elements does not necessarily hold. In practice, this means that the

location of model elements is not determined by their place in the instance-of hierarchy, but instead by other, often unstated, criteria. In other words, in a loose metamodeling hierarchy one simple places model elements at a level where one finds a need to mention them. Although this makes the initial definition of a metamodeling hierarchy much easier, it also gives rise to some subtle, but significant problems [5].

In short, it impacts upon the integrity of the model levels, which effectively end up playing the role of packages that only serve to group elements into subgroups of like purpose. This is not a bad thing in itself, since the value of grouping related model elements within packages has long been established. However, wrapping up what essentially amounts to an application of packages in all the baggage and paraphernalia of "meta" modeling not only becomes confusing, but also is also directly misleading.

A second, and more significant problem is the need to deviate from the well-established mechanism of instantiation in object-oriented approaches to make loose metamodeling work. For example, how does one define a prototypical class (e.g., `Object`) that serves to convey upon entities some basic properties such as that of being an object? The approach used in the specification of the UML (based on loose metamodeling) is to define the prototypical class at the same level of the class from which it is instantiated. The model elements `Class` and `Object` both appear within the (M_2) metamodel, and are related by an unnamed association. But this requires that

1. a modeling element at the M_0 level must be an instance of an M_2 element.
2. a modeling element at the M_0 level must be the direct instance of two classes.

This is clear in the work of Alhir [6], who has to use double, direct instance-of relationships when attempting to fully characterize the relationships between modeling elements within the context of loose metamodeling.

Strict Metamodeling

Strict metamodeling [4] is based on the tenet that if a model B is an instance of another model A then every element of B is an instance-of some element in A (see **Fig. 5**). In other words, it interprets the instance-of relationship between models at the granularity of individual model elements.

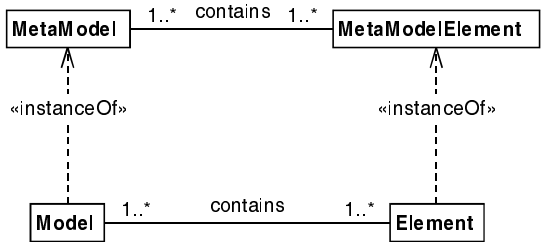


Fig. 5. Strict Metamodeling

The doctrine of strict metamodeling thus holds that the instance-of relationship, and *only* the instance-of relationship, crosses meta-level boundaries, and that every instance-of relationship must cross exactly one meta-level boundary to an immediately adjacent level. This can be captured concisely by the following rule:

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

This definition deliberately rules out the top level of a metamodeling hierarchy, since some way has to be employed to terminate the hierarchy. A common approach is to model the top level so that its elements can be viewed as instances of elements in the same level, i.e., the top-level model is an instance of itself. In essence, the strict metamodeling approach simply seeks to faithfully extend the time-honored separation between the class level and the object level from classic object-oriented development to all levels in multi-level metamodeling architecture.

Although strict metamodeling puts a multi-level metamodeling architecture on a clean foundation [7] at first sight it may seem to complicate what may be perceived as "straightforward modeling". In the following we discuss a number of strategies for solving the problem outlined in the previous section and, in particular, motivate why it is beneficial to detect and resolve complications that only become apparent with a strict metamodeling mindset.

Consistent Process and Product Modeling

After briefly restating the problem observed in the modeling scenario of **Fig. 4** we investigate potential remedies and discuss their respective downsides and virtues.

The Problem

There is a crucial difference between the diagrams in **Fig. 3** and **Fig. 4**: Only in the diagram of **Fig. 4** does an element (`BobsClassDiagram`) at the M_0 level refer to an element (`BobsClass`) at the M_1 level. Element `BobsClass` has deliberately been drawn at the same height as `ClassDiagram` to depict that both are at level M_1. `BobsClassDiagram` is an instance-of an M_1 level element (`ClassDiagram`) and hence resides at the M_0 level. Therefore, according to strict metamodeling, it should not have any relationships with M_1 level elements. There are a number of possible ways around this problem as discussed below.

Avoiding the Combination

One solution is to side step the problem by avoiding the definition of the "process" and "content" aspects of an artifact within the same model. In other words, one solution is to keep the process models and contents models separate by treating products as atomic elements that have no content in a process modeling context. This, however, abandons the idea of a common modeling framework explaining the

relationship between the process and detailed product hierarchies. This is highly unsatisfactory, because it makes it impossible to validate the mutual integrity of the combined process and product hierarchies. Furthermore, the location of a modeling element (`BobsClassDiagram`) within the four-layer metamodeling hierarchy should not depend on its context of use.

We could, alternatively, accept that `BobsClassDiagram` has to appear twice as two unrelated concepts and assign each occurrence to a different level. This has to be considered as escaping the real problem, since both are clearly referring to a single real class diagram. The best interpretation of such an approach is probably to declare the occurrence within the UML definition as the defining one and treat the occurrence within the process definition as a use-reference. This is not entirely satisfying either because there is no well-defined relationship between a defining occurrence and a use-reference especially if these are at different metalevels. More importantly, we would allowing a single real thing to be represented by two ostensibly unrelated concepts for no other reason than to try to resolve what should be a natural modeling scenario.

Adopting Loose Metamodeling

Another solution is to turn a blind eye to the proper location of complex artifacts, such as class diagrams, and to pick one of the levels arbitrarily. One then, of course, needs to adopt a loose metamodeling approach because relationships, such as associations, will then need to cross meta-level boundaries. The last section of this paper discussed why such an approach is ill-defined without a proper interpretation of such level crossings. An ideal solution would identify the one proper level for a single `BobsClassDiagram` and then accommodate this level consistently within both the process and the product hierarchy.

Elevating the Process Hierarchy

Avoiding the level crossing of the association between `BobsClassDiagram` and `BobsClass` can easily be avoided by shifting the whole process hierarchy (see **Fig. 6**) up one level. From a pure metamodeling perspective the *absolute* position of elements is irrelevant. There is no fundamental reason to have instances (e.g., `BobDesigns`) only at the M_0 level. The fact that `BobDesigns` is not further instantiable can be expressed at any level (e.g., by underlining its name). It is therefore possible to promote it from M_0 to M_1 and with it all the other elements of the process hierarchy respectively. This would seem to have restored strictness and, provided that we could model within an unlimited metamodeling hierarchy, all the relationships would seem to make sense and be sound. However, within the context of the limited four-layer-modeling hierarchy we have just "pushed" elements `Activity` and `Diagram` into the M_3 level, which is supposed to be reserved for the MOF to define how modeling at the M_2 level works and not for accommodating promoted M_2 elements.

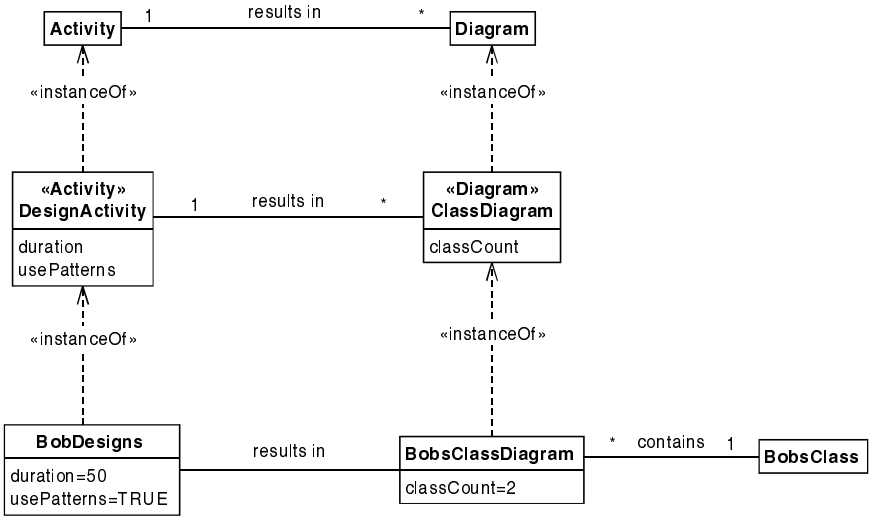


Fig. 6. Shifting Up the Process Hierarchy

Interestingly, this problem hints at the correct position and purpose of the elements `Activity` and `Diagram`. If we replaced the instance-of relationships between the M_2 and M_3 levels with inheritance links within the M_2 level – thereby moving Activity and Diagram down to the M_2 level – we would avoid promoting modeling elements into the M_3 level. To understand the subtleties of this strategy we first need to investigate the different characteristics of instantiation versus inheritance.

Instantiation versus Inheritance

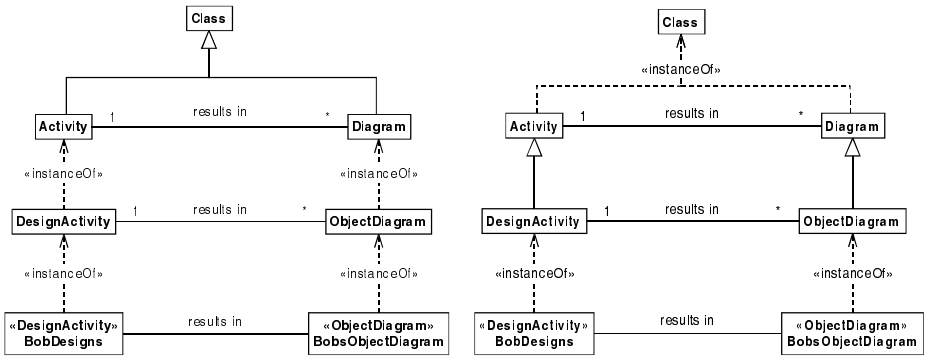
Although instantiation and inheritance are quite different mechanisms they can often be used to serve the same purpose. They both can be used to determine the shape of derived elements from existing elements. For example, in order to define what a `DesignActivity` is (i.e. give it a type and some predefined features) one can –

- *classify* what it is by providing a definition of its class (i.e., use instantiation), or
- define it as a *specialization* by providing a definition of its superclass (i.e., usinheritance).

Although the mechanisms seem similar[1] in what they can achieve in terms of defining the properties of derived elements, one has to be aware of a number of fundamental differences. For the purpose of the following discussion we are now assuming that the activity `BobDesigns` is to produce `BobsObjectDiagram` rather than `BobsClassDiagram`. Object diagrams contain M_0 level objects and, thus, should be placed at the M_0 level. The advantage of assuming that `BobDesigns` produces something at the M_0 level is that we can take **Fig. 4** as a starting point without the

[1] Both relationships may be very imprecisely referred to as "is-a", e.g., `Bob` is-a `Person` and `Student` is-a `Person`, where `Bob` is an instance and `Student` a subclass of `Person`.

need to first resolve the location of `BobsClassDiagram`. In this way we can directly compare two alternative modeling scenarios (see **Fig. 7**) with all elements being in the level where one would usually expect them without worrying about shifting hierarchies, etc.



(a) Metamodeling (b) Modeling

Fig. 7. Comparing Metamodeling to Modeling

The first modeling instinct when defining a new vocabulary, such as a language to describe processes, seems to be to create a meta-level and describe new concepts as Metaclasses, i.e., one performs metamodeling. This explains why there is an element `Activity` at the M_2 level in the original process/product diagram (see **Fig. 4**). Even though it does not provide any attributes or methods, it is provided as generator class for instances such as `DesignActivity`.

Attribute Transitivity

In fact, any definition of attributes or methods in element `Activity` at the M_2 level would hardly make any sense. After all, the purpose of modeling is to describe the properties of instances (usually at the M_0 level)[2]. If we had an attribute `duration` in the M_2 level element `Activity` then `DesignActivity` would receive a slot `duration`. An instance of `DesignActivity`, e.g., `BobDesigns`, however, would not. `BobDesigns` only receives slots as described in the form of attributes in element `DesignActivity`. Attributes defined at the M_2 element `Activity` only make sense to convey class level information (in this case for `DesignActivity`), such as the author's name or version information about the class. Obviously, in an instantiation hierarchy the effect range of attributes has chain length one, since attributes become slots in the instantiated element and then are of no relevance for instances further down the chain. In other words, in terms of attribute propagation the instantiation mechanism is not transitive.

The inheritance mechanism, however, is transitive in terms of attribute propagation. Even in an arbitrarily long inheritance chain the bottom-most class will still receive all of the attributes of the top-most class, i.e., the effect range of attributes

[2] **Fig. 6** shows a class with attributes at the M_2 level which only makes sense in this special case because the instance to be defined are at the M_1 level here, i.e., there are no M_0 instances to be shaped.

has infinite chain length [3]. Similar to attributes, associations are only propagated to derived elements in the case of instantiation. Modeling relationships at the M_2 level is thus irrelevant for the shape of M_0 elements. For instance, the "`results in`" relationship between `BobDesigns` and `BobsObjectDiagram` in **Fig. 7** (a) has not been created because of the relationship between `Activity` and `Diagram` as the latter only creates a link between `DesignActivity` and `ClassDiagram` and not the association visible in the diagram. In **Fig. 7** (b), though, the relationship between `Activity` and `Diagram` *is* responsible for the link between `BobDesigns` and `BobsObjectDiagram`.

Polymorphism

By the same token, the "member-of"[3] relation works transitively only for inheritance chains, e.g., `BobDesigns` is a member-of `Activity`, because it is a direct "instance-of" `DesignActivity` and an indirect instance of `Activity` assuming the scenario in **Fig. 7** (b).

In other words, inheritance enables polymorphism, that is, instances can be viewed as being instances of superclasses as well. Polymorphism is one of the main features of object-oriented languages that facilitates the creation of extensible software. In a modeling context, polymorphism could be exploited to define constraints for a number of classes only once at their common inheritance root. Also, using polymorphism one can delegate certain checks to element types rather than using case analysis over the state of an element instance.

Individualization

Instantiation, on the other hand, is required when the element to be defined must carry its individual state. `BobsObjectDiagram`, for instance, can never be defined by use of inheritance from `ObjectDiagram` since it must maintain its individual value for `classCount`.

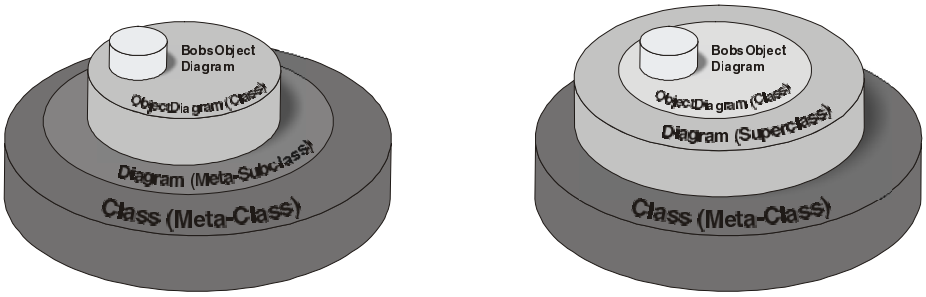
Before we come back to resolving the original problem of achieving a satisfying modeling scenario for the elements contained in **Fig. 4**, we will briefly look at the differences between inheritance and instantiation from a slightly different angle along with guidelines for how to choose between the two.

Specialization versus Classification

The two mechanisms we have analyzed for defining derived modeling elements – instantiation and inheritance – are basically object-oriented mechanisms for realizing the modeling concepts of classification and specialization respectively. How can we confidently decide which is more appropriate?

Fig. 8 gives another rendering of the product hierarchy of the modeling alternatives depicted in **Fig. 7**.

[3] An element is a "member-of" another element if it is either a direct or indirect "instance-of" of the other element.



(a) `ObjectDiagram` "instance-of" `Diagram`

(b) `ObjectDiagram` subclass of `Diagram`

Fig. 8. Classification versus Specialization

In both subfigures **Fig. 8** (a) and **Fig. 8** (b), the instance `BobsObjectDiagram` is depicted as the topmost white cylinder. Note that cylinders indicate instantiation while subcircles indicate subtyping, i.e., inheritance. One can clearly see that the scenarios are very similar – especially the bottom-most and top-most elements – but there is one essential difference: In **Fig. 8** (a) `Diagram` is a Metaclass, i.e., plays the role of a class for `ObjectDiagram`, whereas `Diagram` is a class in **Fig. 8** (b), i.e., plays the role of a superclass for `ObjectDiagram`. Hence, the question as to whether to use classification (**Fig. 8** (a)) or specialization (**Fig. 8** (b)) as a way of defining `ObjectDiagram` can be rephrased as: What is the proper role of `Diagram`? Is it a Metaclass or a class? Do we need to metamodel or simply model? Note that apparently it does not seem to matter much for instance `BobsObjectDiagram`, as either first specialization and then classification (in subfigure **Fig. 8** (a)) or first classification and then specialization can be used (in subfigure **Fig. 8** (b)).

In general, metamodeling is only required if a new concept (e.g., `Diagram`) cannot be expressed as an instance of an existing Metaclass. This is the case, for instance, with links between objects, which are fundamentally different to objects so that they warrant a different Metaclass, i.e., not `Class` but `Association`. If an existing Metaclass is sufficient to determine what a new concept (e.g., `ObjectDiagram`) essentially is, then it is appropriate to let the new concept be an instance of that Metaclass typically by additionally creating a superclass (e.g., `Diagram`) which generalizes upon the properties of the new concept. In our example, for all intents and purposes `ObjectDiagrams` are to be treated as classes in terms of their modeling properties, so that the scenario in **Fig. 8** (b) is appropriate.

This general guideline, although valid, is unfortunately not very concrete. However, a full understanding of the differences between instantiation and inheritance now allows the questions about the appropriate level for `Diagram` to be cast in the following more simple form –

1. *Do we want attributes in `Diagram` (e.g., `duration`) to become slots in `BobsObjectDiagram`?*
   If yes, then `Diagram` needs to go to level M_1.

2. *Do we want instances of* `ObjectDiagram` *(e.g.,* ***`BobsObjectDiagram`****) to be members-of* ***`Diagram`****?* This is useful if we specify constraints at `Diagram` which should hold for instances such as `BobsObjectDiagram`.
   If yes, then `Diagram` needs to go to level M_1.
3. *Do we need* ***`ObjectDiagram`*** *to have individual state with regard to* `Diagram`*?* For instance, `BobsObjectDiagram` and `JoesObjectDiagram` need individual state (`classCount`) with regard to `ObjectDiagram`[4].
   If yes, then `Diagram` needs to go to M_2 assuming that `ObjectDiagram` is at M_1.

Answering these questions makes it obvious that `Diagram` has to go to level M_1 and thus that the scenario in **Fig. 8** (b) is more appropriate. Clearly, it is easier to justify the individualization from `Class` to `Diagram` in **Fig. 8** (b) as opposed to the individualization from `Diagram` to `ObjectDiagram` in **Fig. 8** (a). Also, one would assume that `BobsObjectDiagram` is a member-of `Diagram`. For example a guideline stating that diagrams may not contain more than 20 elements may be formulated in element `Diagram`. This is only the case in **Fig. 8** (b). If the assumption were wrong then this would be an indication that `Diagram` is misnamed!

When attempting to express the same constraint in the scenario shown in **Fig. 8** (a) one either needs to repeat this for every kind of diagram (e.g., `ClassDiagram`, `ObjectDiagram`, etc.) or, within `Diagram` one needs to talk about the instances of the instances of `Diagram` which clearly is not a straightforward way of modeling properties.

Now that we have established a decision process for determining whether an element should play the role of a Metaclass (at M_2) or should have the role of a superclass (at M_1), we are now in a position to providing a more satisfying modeling scenario for the elements contained in **Fig. 4**.

Removing a Metamodeling Level

Using the scenario shown in **Fig. 6** as a starting point, one can actually arrive at a modeling scenario that is fully consistent and perfectly fits into the four-layer metamodeling hierarchy. We simply implement the result of the analysis of the previous section and replace the instance-of relationships between M_2 and M_3 elements with inheritance links, thereby moving elements `Activity` and `Diagram` from level M_3 to level M_2 (see **Fig. 9**).

[4] This is why objects are always an instance-of some class and never inherit from another object. Inheriting from another object only makes sense if the state of the "superobject" is to be shared among several "subobjects". Only prototype-based languages enable such a configuration.

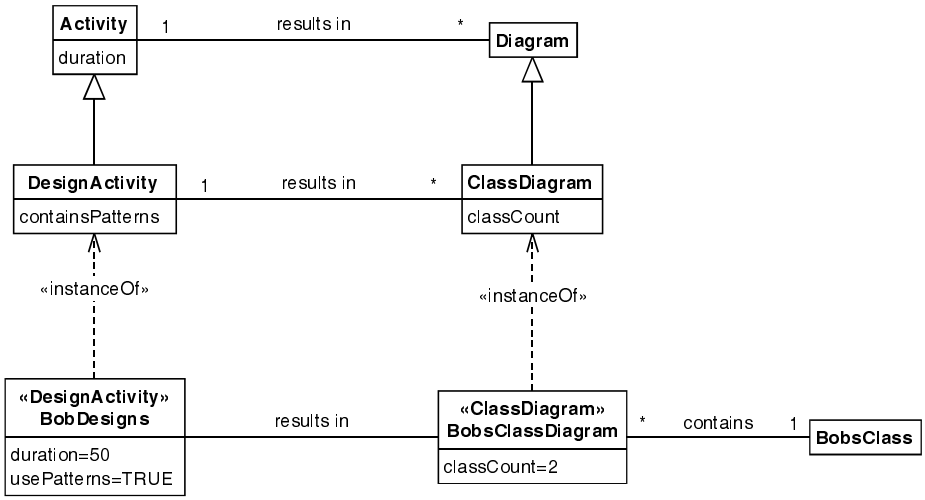


Fig. 9. Strictness restored

In comparison to **Fig. 6** one can identify a number of differences –

- elements `Activity` and `Diagram` are now at the M_2 level.
- the instance-of relationships formerly existing between levels M_3 and M_2 have been replaced with inheritance links within level M_2.
- element `Activity` now defines an attribute `duration`.
- element `DesignActivity` no longer defines an attribute `duration`, but inherits it from `Activity`.
- the M_0 level did not change.

The last statement states an important fact: The things we are modeling, i.e., process enactments (`BobDesigns`) and products of a design phase (`BobsClassDiagram`), still have the same shape. The nature of the things we are trying to capture with our modeling efforts has not been affected by our refactoring of the metamodeling structure. As for the properties of the defining metamodeling structure the refactoring can be fully justified, using information from the previous analysis about the difference between classification and specialization. With regard to *attribute transitivity* it is now possible – by virtue of the inheritance link between `Activiy` and `DesignActivity` – to perform the modeling optimization already addressed in the second section of this paper. The fact that all activities have a `duration` can now be properly expressed by defining `duration` within `Activity` rather than `DesignActivity` (see **Fig. 9**). If not all activities should posses a duration feature than we may opt to define it in `DesignActivity` but at least we now have the choice.

With regard to polymorphism, it is now possible to regard design enactments (e.g., BobDesigns) as members-of Activity. This is beneficial if one wants to make statements about activities in general (not just design activities), which should hold for activity instances. Note that Activity and Diagram were really misnomers before (see Fig. 4). Previously an instance of, e.g., Activity was not an activity instance but an Activity class. Now (indirect) instances of Activity are true activity instances.

With regard to *individualization*, it seems much more natural to justify the individualization from `Class` to `Activity` in **Fig. 7** (a) compared to the individualization from `Activity` to `DesignActivity` in **Fig. 7** (b).

In summary, the refactoring of the metamodeling structure, which was motivated by observing a strictness violation, led to a modeling scenario that seems to be much more faithful to the underlying concepts.

A Modeling Pattern

The modeling scenario presented in section "Removing a Metamodeling Level" not only has a better logical foundation but also exhibits better modeling properties. In fact, the case for modeling at the M_1 level (as opposed to metamodeling at the M_2 level) is such a strong one that we expect to see this structure recur in several other modeling contexts as well. Note that *where*, within the four-layer metamodeling hierarchy this structure is applied (i.e., the absolute level where it can be found) is not as important as its internal structure. With the class diagram and object diagram design processes we have already shown that the structure occurs at both the M_2/M_1 and M_1/M_0 levels. The identifying property of this structure, which we would like to label the "Prototypical Concept Pattern[5]" is that, first, a general prototypical concept is specialized into a more refined concept and then an instance-of relationship yields an instance which has the combined properties of both concepts and can be considered a member-of the prototypical concept. Other examples of this pattern within the UML are the `Object` root of all classes and the `Link` root for all associations.

Metacycles, Metabombs and Metamodeling Spaces

Repairing the initially non-strict model of **Fig. 4** with a structure (see **Fig. 9**) that later on was found to be a general solution for a number of such problems may create the illusion that strict models may be obtained with this pattern all the time. Unfortunately, we can find examples where strictness cannot be restored so easily. Before we proceed to these harder examples we introduce the notions of metacyles and metabombs.

Metacycles

Fig. 10 shows the general abstract nature of a non-strict model. The `refers-to` association crosses a meta-level boundary and thus violates strictness. Why is this a problem and why does it need a special term? First, as one can observe in **Fig. 10**, `Instance` depends on `ModelElement` (through the instantiation relationship) but `ModelElement` (through the directed association) also depends on `Instance`.

[5] The context in which the problem solved by this structure may appear and the resulting properties can be well described in pattern form. As the pattern may occur at any level it is more appropriate to refer to "concept" rather than "class" or "object".

This introduces a dependency cycle that one would like to avoid between classes even in regular modeling, let alone between two elements which are supposed to be at two different modeling levels.

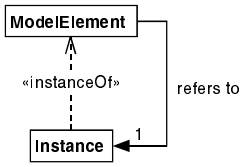


Fig. 10. A Metacycle

Let us think of the instances as representing games of chess and the modeling level as describing the rules of playing chess [8]. **Fig. 10** expresses the fact that the rules of chess depend on an actual game of chess, which is clearly counterintuitive at the least. Although chess rules need to describe scenarios of chess piece configurations and need to refer to game history this is entirely different to assuming that with one actual game one is changing the rules of chess. Clearly one assumes the rules of chess to be universal and constant for all games without being distorted by certain game instances. The same standard of a clean separation between definition and enactment should be carried out in modeling within a multi-layer modeling hierarchy.

Metabombs

In fact, metacycles are even worse than they have just been characterized. **Fig. 11** shows a strictness violation, which appears to be less harmful, but is in fact as fatal in its consequences as a metacycle. At first sight the situation looks less problematic then in **Fig. 10** since the dependencies point in one direction only. In the chess analogy, games would make references to the rules, which normally is not necessary as the rules are already part of games by means of instantiation.

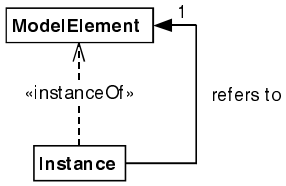


Fig. 11. A Metabomb

However, let us assume for the moment that rules may be changed dynamically. Let us imagine that in the middle of the world championship the maximum contemplation time for one move is limited because one of the opponents takes the liberty of thinking whole days about a single move.

Ordinary chess game instances (without a reference to their class) can not be notified of the change. Once they are instantiated they "lead their own life" independently of their generating class. Those chess games, however, which may

consult the rules defined in the generating class dynamically, could make immediate use of a rule change. Not surprisingly there are programming languages (such as Smalltalk and Java), which allow instances to access classes for information regarding class variable values or the availability of methods. So why is there a problem with regard to metamodeling in a multi-level hierarchy?

The answer can be illuminated by posing a simple question: Is the line labeled `refers-to` a link or an association? Objects (`Instance`) are connected by links whereas classes (`ModelElement`) are connected by associations. What is the nature of a line connecting an object with a class, though?

Obviously, this line is of a third kind whose nature has to be described in a metalevel relative to the level we are looking at. But what is the metalevel to a scenario mentioning elements from both the M_0 and M_1 levels? The M_2 level only models the M_1 level and should not be concerned with instances of the model it is describing. If, in the M_2 level, one finds a need to mention M_0 elements it means only one thing, that the M_1 and M_0 levels in reality are just one big level. Any metalevel attempting to describe the semantics of **Fig. 11** must regard it as a single level from its perspective. By the same token, modeling the semantics of programming languages that allow instances to reference their classes implies that the instances and classes are in the same modeling level. One may find a use for having a substructure within that one level to distinguish the internal class and the object levels, but from the perspective of the defining modeling level this substructure is irrelevant because relations from the defining level have to access all substructure levels to define the semantics of their interrelations. This is why a structure of the form depicted in **Fig. 11** or in **Fig. 10** or in general any non-strict modeling scenario can be called a metabomb. Any non-strict relationship collapses the two levels it connects into a single level, since the meaning of the relationship can only be defined at a further metalevel which must assume the two connected levels to be a single level only. In fact, there must always be a conceptual metalevel to the whole mutli-level hierarchy giving meaning to the instance-of relationship between the levels.

Now that we have seen that strictness violations are not a mere faux pas but are a fundamental threat to the integrity of any multi-level hierarchy we will have a look at examples which are not as easy to cure as the ones we have seen previously. After that we discuss a conceptual remedy for scenarios which do not, by their inherent nature, fit into a single meta-model hierarchy.

Hard Metacycles

We have managed to remove the non-strictness in **Fig. 4** by shifting some classes and applying the Prototypical Concept Pattern. However, **Fig. 12** shows a modeling scenario that cannot be cured with this approach. In this scenario the creator of the model wishes to express the fact that Bob, in his design activity, produces object diagrams as well as class diagrams.

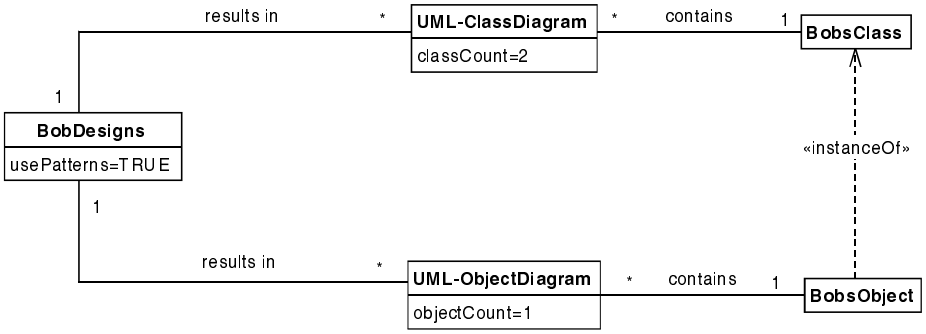


Fig. 12. A hard metacycle

This is quite a valid scenario to model but introduce a strictness violation: `BobDesigns` references elements from the M_1 level (`ClassDiagram`) and the M_0 level (`ObjectDiagram`). The position of these two diagram types is fixed by their respective reference to elements that unequivocally belong to the M_1 (classes) and M_0 (objects) levels. This first raises the question as to which level `BobDesigns` should go and, second, introduces a non-strict relationship in either case. Potentially, one could resolve this problem by duplicating or rather splitting `BobDesigns` to two levels (see **Fig. 13**).

What at first appears to be a crude fix may even make sense in this case because designing a class diagram may require different activity properties and or activity states than designing an object diagram. However, somewhere the two `BobDesigns` abstractions should join in a common model if one attempts to describe the process framework as a coherent whole rather than split over two metamodeling levels.

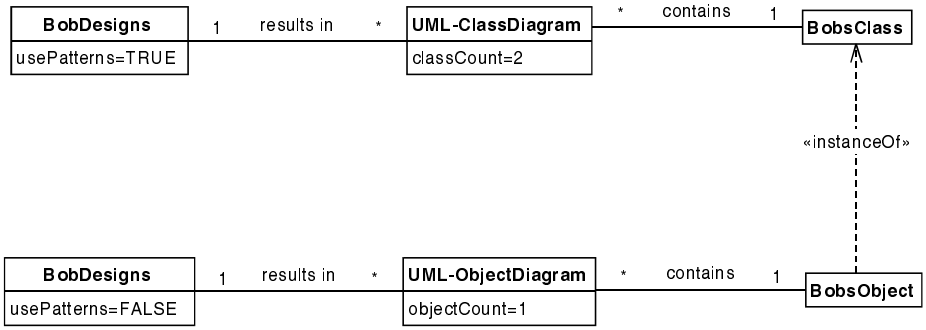


Fig. 13. Repeating Structures to maintain Strictness

For example, an originating `Project` instance is presumably expected to be located at one level only. It furthermore does not seem to be elegant to repeat large parts of the process model simply to avoid strictness violations. Even worse, this duplication could occur again, e.g., due to the fact that Bob starts to create artifacts at the M_2 level as well. Thus, while a duplication/splitting approach seems viable for partial modeling activities a more general solution is required in general.

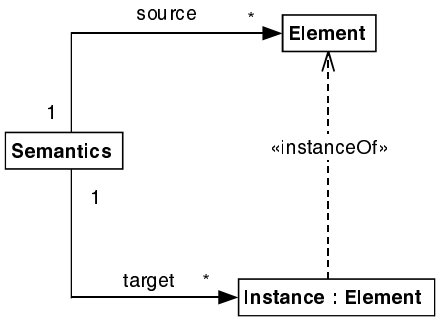


Fig. 14. A Really Hard Metacycle

Fig. 14 shows a situation similar to **Fig. 12**. In this case an element `Semantics` needs to access both a modeling element (`Element`) and its instance (`Instance`) in order to define the semantic relationship between them. Again, at first it is not clear at which level `Semantics` should go but from the discussion in section "A Metabomb" it is quite clear that `Semantics` needs to be placed at a metalevel relative to both `Element` and `Instance` which in turn – from `Semantics`' perspective – reside at a single modeling level. Is it, nevertheless, possible to follow a strict metamodeling approach in such cases? The following section presents a view that reconciles strictness requirements between modeling levels and the existence of abstractions that need to mention multiple modeling levels at the same time.

Metamodeling Spaces

The key insight for resolving the conflict of seemingly unavoidable strictness violations as presented above is to recognize that fitting all these modeling domains into a single four-layer modeling hierarchy is impossible without abandoning strictness and, hence, the multiple metalevels themselves!

The above observation is a simple consequence of the fact that some modeling domains make statements about other modeling domains. In particular, processes mention products and semantic definitions mention elements from multiple metalevels. The key to accommodating all these domains into a single hierarchy is to separate the domains into metamodeling spaces. As a result we obtain (a non exhaustive list of) three metamodeling spaces, one containing UML elements, one containing process modeling elements, and one containing semantic definitions. **Fig. 15** shows a process modeling space to the left (with levels P_0-P_2) and a UML modeling space to the right with levels (M_0-M_2). Strictness can now be enforced both locally (within metamodeling spaces) and globally (between the metamodeling spaces). The one source of a strictness violation – the reference of a design activity instance to a diagram class (see **Fig. 4**) – has been taken outside each of the separate modeling spaces, which makes them locally strict.

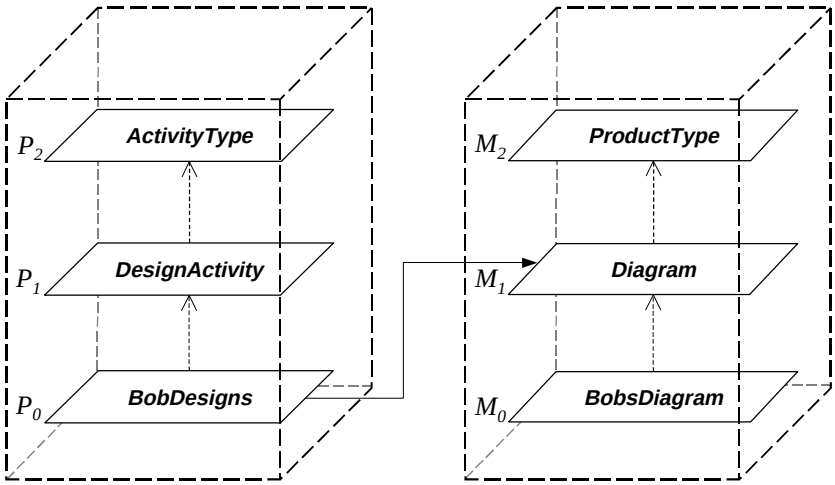


Fig. 15: Strict references across Metamodeling Spaces

The global reference from `BobDesigns` to `Diagram` now takes place between two modeling spaces and as such need not respect internal level cardinalities[6]. The fact that this reference originates at the P_0 level and targets an M_1 level does not constitute a strictness violation. This can readily be seen by the distinct letters. Any level within the P metamodeling space is independent from any level within the M metamodeling space. The important observation to make is that the M metamodeling space *as a whole* is subordinate to the P metamodeling space. The P metamodeling space is "meta" to the M metamodeling space but in a dimension (e.g., horizontal) different to the dimension (e.g., vertical) used within the metamodeling spaces. This implies that global references between metamodeling spaces are of a different nature to the associations within the metamodeling spaces. Their precise meaning has to be defined outside any of the metamodeling spaces.

It needs to be understood that the metamodeling spaces form their own hierarchy, which is independent and outside the traditional multi-level-hierarchy used inside the metamodeling spaces. Conceptually, the P metamodeling space is on a meta-level relative to the M metamodeling space. However, this meta-level dimension is a different one compared to the meta-dimension within the metamodeling spaces and, also, we say "conceptually" because the M metamodeling space is not an instance-of the P metamodeling space. The conceptual meta-relationship between metamodeling spaces expresses a dependency direction, i.e., one metamodeling space (e.g., M) is the target of another one (e.g., P), and also indicates that one metamodeling space (e.g., P) can see all of the levels within the target (e.g., M) at the same time. As a consequence, all potential for a cyclic definition (referred to as metacylcles or metabombs before) has been removed. Coming back to the chess analogy we now have a modeling architecture where rules can be strictly separated from games *and* where it is possible to talk about the relationship between rules and games if

[6] Note that the solution shown in **Fig. 9** is still superior to this scenario, which we simply use to demonstrate that another way to restore strictness is possible as well.

necessary, all in a manner with clear dependencies, ruling out unintended collapsing of metalevels and unwanted cyclic definitions.

An Interpretation of Mixed Space Diagrams

Conceptualizing metamodeling spaces provided a solution to the seemingly unavoidable strictness violations we have encountered in the modeling of processes and semantic definitions. Do we always have to draw three-dimensional diagrams now in order to faithfully respect metamodeling spaces? Is it illegal to draw a diagram such as the one in **Fig. 14**? The most practical approach would appear to be that such diagrams are to be admitted as long as there is a clear understanding that they represent a projection of two metamodeling spaces onto one diagram. Just as we are happy to accept a certain configuration of polygons to be a two dimensional representation of a three dimensional cube, we may accept such projected diagrams as seen in **Fig. 4**, **Fig. 12** or **Fig. 14**. It has to be clear, though, that these present a shortcut notation to a modeling architecture with clean dependencies within- and between metamodeling spaces.

Conclusion

The object-oriented modeling of software processes, as well as products, is receiving growing attention in the software engineering industry (e.g., OMG process metamodel). When the products of a process are treated as primitive entities, the initially appealing approach of defining the basic concept of a process (or activity) and a product at the metamodel level (i.e., M2) seems to work reasonably well. The only minor irritation is that attributes of the process entity at the M2 level are not conferred upon actual process enactments at the M0 level (after two instantiations).

However, when the products of a process are treated as complex entities (e.g. UML diagrams), and the contents of these products are modeled within the same multi-level modeling framework as the process, more troublesome difficulties arise. The basic problem is that the "natural" level for process instances does not always match the "natural" level for the product instances that they create. Assuming one wishes to capture the fact that the process creates the products, this forces associations to cross meta-level boundaries and thus violate the doctrines of strict metamodeling.

Although the disciplines of strict metamodeling might initially complicate the creation of a hierarchy of metamodels, the increased rigor and consistency that it enforces make the return on investment well worthwhile (much like strong typing for programming languages). Therefore, its benefits should not be given up lightly, even if an initially appealing justification is found for violating strictness and any attempts to avoid the violation at first lead to further complications. Fortunately, help was found in perhaps an unlikely quarter - the traditional inheritance mechanism, which distinguishes object-oriented from object-based approaches.

Close analysis of the properties of inheritance and instantiation reveals that they have some surprisingly similar but also some very distinct properties. They both offer a way of capturing what at first sight seem to be forms of "is a" relationships. However, the differences in their detailed semantics means that they are suited to

different mutually exclusive "is a" scenarios. In the process modeling scenario, careful analysis of the respective semantics reveals that inheritance is more suited than instantiation for the relationship between the general process concept and specific types of process. The same situation holds also for products. When inheritance is used instead of instantiation in this situation, the relative levels of the products and process changes, and the level mismatch that initially appeared to force a strictness violation disappears. As a side effect, the minor problem with attribute transitivity also disappears.

The interplay between inheritance and instantiation that resolved this approach seems to be helpful in numerous metamodeling situations. We therefore developed a generalization of this strategy, which qualifies as a potential pattern for multi-level metamodeling scenarios. The pattern, referred to as the Prototypical Concept Pattern, works by placing the prototypical form of an entity at the root of an inheritance hierarchy at the same level as more specialized examples, rather than as a meta class at the meta level above. The resulting structure appears somewhat similar to the concepts of powertypes [9].

Although this pattern resolves the problem when a process creates a product at one level only, it seems unreasonable to rule out the possibility that a single process might also create multiple products at different levels. This problem is not solvable by clever patterns of inheritance and instantiation, but requires a more complex approach. To handle such situations while remaining faithful to the ideas of strict metamodeling, it is necessary to understand that there are some areas of concern (e.g., processes, semantics) which regard other areas of concern (e.g., UML products, UML entities) as single entities. In other words, they disregard any level boundaries in their target area of concern. Inevitably, such areas of concern are "meta" to their target areas of concern. We have outlined an approach which involves arranging multiple metamodeling hierarchies (metaspaces) in such a way that there can be dependencies between them. Another approach is to stay within one hierarchy only and use the next level higher up to define dependencies between areas of concern at a particular level. As one can easily imagine, the latter approach seems to lead to a more convoluted overall scenario. A thorough comparison of the two approaches is, however, outside the scope of this paper.

Ironically, when one projects two modeling hierarchies (metaspaces) and the dependencies between them into a single diagram, models depicted in such a way can actually have what appear to be level crossing associations. Provided that the true semantics of such relationships are understood, however, such diagrams can be used as a shorthand notation for the real underlying model. The proposed multi-dimension modeling hierarchies approach therefore represents the most general form of modeling framework, and is sufficiently powerful to handle all conceivable modeling scenarios in a way that conforms to the spirit of strict metamodeling.

Acknowledgments

The authors would like to thank Jörn Bohling, Dirk Riehle, Paul Sammut, Brian Henderson-Sellers, and the anonymous reviewers for their helpful comments and Cris Kobryn for his characterization of strict metamodeling in the form of a diagram.

References

1. OMG: OMG Unified Modeling Language Specification, Version 1.3., OMG document ad/99-06-08 (1999)
2. D. Firesmith, B. Henderson-Sellers, and I. Graham, *OPEN Modeling Language: (OML) Reference Manual*, Cambridge University Press, 1998.
3. Atkinson C., Kühne T. and Henderson-Sellers B.: To Meta or not to Meta – That is the Question. Journal of Object-Oriented Programming, December (2000) 32-35
4. Atkinson, C.: Supporting and Applying the UML Conceptual Framework. In: UML'98 (1998)
5. Atkinson C., Kühne T.: Strict Profiles: Why and How. In: Proceedings of UML 2000, LNCS 1939, October (2000) 309-322
6. Alhir, S., S.: Extending the Unified Modeling Language. At: home.earthlink.net/~salhir (1999)
7. Kobryn, C.: UML 2001: A Standardization Odyssey. Communications of the ACM (42):10 (1999) 29–37
8. Hofstadter, D.R.: Gödel, Escher, Bach: An Eternal Golden Braid, Basic Books (1979)
9. Odell, J.: Power Types, Journal of Object-Oriented Programming, May (1994)