Softw Syst Model
DOI 10.1007/s10270-007-0061-0

REGULAR PAPER

Reducing accidental complexity in domain models

Colin Atkinson · Thomas Kühne

Received: 22 December 2006 / Revised: 19 April 2007 / Accepted: 3 May 2007

Abstract A fundamental principle in engineering, including software engineering, is to minimize the amount of accidental complexity which is introduced into engineering solutions due to mismatches between a problem and the technology used to represent the problem. As model-driven development moves to the center stage of software engineering, it is particularly important that this principle be applied to the technologies used to create and manipulate models, especially models that are intended to be free of solution decisions. At present, however, there is a significant mismatch between the "two level" modeling paradigm used to construct mainstream domain models and the conceptual information such models are required to represent—a mismatch that makes such models more complex than they need be. In this paper, we identify the precise nature of the mismatch, discuss a number of more or less satisfactory workarounds, and show how it can be avoided.

Keywords Domain modeling · Model quality · Accidental complexity · Modeling languages · Modeling paradigm · Stereotypes · Powertypes · Deep instantiation

Communicated by Professor Bernhard Rumpe.

C. Atkinson
University of Mannheim, Mannheim, Germany
e-mail: atkinson@informatik.uni-mannheim.de

T. Kühne (✉)
Darmstadt University of Technology,
Darmstadt, Germany
e-mail: kuehne@informatik.tu-darmstadt.de

1 Introduction

Whether performed through traditional software engineering activities or advanced model-transformation techniques, the development of a new software system usually involves the construction of a solution-independent description of the problem to be solved. Such models are referred to as "domain models" [18] or "analysis models" [8]. As these names imply, the goal is to make such models as free as possible from considerations related to specific solutions or solution technologies so as to not embody any premature decisions that may hamper later development.

One important purpose of domain models is to serve as a description of the problem that is understandable to the widest possible range of stakeholders. In order to determine whether an agreement about the system requirements has been achieved and whether the system domain has been accurately captured, it is highly desirable that even people without a computer science background be able to at least partially understand a domain model.

Note that in the context of this paper, a domain model is assumed to represent a conceptualization of the entities of some problem domain. Our "universe of discourse" is therefore not a set of physical entities from the real world but contains elements that were created in a conceptualization, e.g., by domain analysts. Measuring the adequacy of domain models with respect to a conceptualization, as opposed to the real world is useful in order to avoid philosophical arguments of whether universals [22] should be assumed to exist in the real world and what constitutes an appropriate model of an excerpt of the real world.

The most common choice for creating such domain models is the modeling language standard, the UML [27]. Like many of its predecessors and competitors, the UML is based on the ubiquitous object-oriented paradigm. As a

result, creating a domain model with the UML usually includes the creation of one or more class diagrams that capture the important domain concepts and their relationships. Often one also adds one or more object diagrams showing instances of the domain concepts to illustrate various configurations. In the implementation phase, the elements of the class diagram(s) can then be more or less straightforwardly mapped to classes of an object-oriented programming language via an intermediate design phase.

Creating domain models using the same underlying paradigm as the ultimate implementation technologies is advantageous in terms of a seamless transition from problem to solution. However, it may stand in tension with the goal of completely separating the problem description from the solution technologies. If the solution paradigm tarnishes or complicates the representation of concepts within domain models, the disadvantages of having domain models polluted with solution restrictions may outweigh the advantages of a completely seamless development process. The overriding goal of domain modeling must, by definition, be to represent the problem space concepts in as faithful and untarnished a way as possible. Anything that stands in the way of this goal leads to suboptimal domain models with immediate effects on their primary purposes of being stable against the change of solution technologies and being communication vehicles between experts and non-experts alike.

The main objective of this article is to make the case that the object-oriented paradigm currently underpinning the UML and other standards does not fulfill this goal, and as a consequence adds unnecessary (i.e., accidental) complexity to domain models. The object paradigm per se is not the problem but a rather unnecessary, historically motivated restriction to modeling at two levels only. The UML supports the above mentioned two levels very well, e.g., in the form of object diagrams (instance level) and class diagrams (type level), but provides only meager support for further levels. Consequently, if one needs to create a model of a domain involving more than two levels using a two-level language like the UML, one is forced to use artificial workaround mechanisms or modeling patterns that allow the properties of multi-level scenarios to be mimicked using only two levels. Typical examples addressing this need include static variables, tagged values, stereotypes, powertypes, reflection, and a number of variations of the "Item Description" pattern [7]. The problem with such workarounds is that they complicate and obscure the meaning of a domain model. Each workaround comes with its own set of idiosyncrasies, often making it difficult for uninitiated readers of the model to understand which aspects of the model are meant to be *accurate reflections* of the *problem domain* and which are just *accidental properties* of the particular *workaround*.

We argue that the use of such "workaround" techniques is suboptimal and increases the accidental complexity of domain models. We use the term "accidental complexity" in the sense of [6] and assume that if one model conveys the same information as another model, but in a more concise way using less modeling elements and concepts, it is less complex. We therefore argue that models of domains that inherently require more than two modeling levels should use a modeling approach and infrastructure that explicitly and cleanly supports more than two domain modeling levels. The paper makes two basic contributions. The first is to properly characterize the nature of the problem that occurs when trying to model domains with more than two logical metalevels using languages that support only two. The second is to compare the strengths and weaknesses of potential solutions addressing this problem.

Note that although we discuss the above mentioned issues in relation to the UML, the fundamental problem and suggested solution are not unique to the UML. On the contrary, through the provision of powertypes and its extension mechanism the UML provides better support than most modeling languages for capturing problems that inherently contain more than two modeling levels. The basic problem and solution discussed in this article are therefore of relevance to all modeling languages based on the two-level, object-oriented paradigm.

In the next section we introduce a small example and then, in Sect. 3, demonstrate a number of typical workaround practices used to represent multi-level domain models using only two-levels. In Sect. 4, we proceed to discuss attempts to provide enhanced support for more than two levels in the modeling language. Section 5 then shows how the same domain information can be captured more concisely and precisely using a multi-level modeling paradigm and finally extends the discussion to models where multi-level modeling is more than just a sequence of two-level modeling pairs, i.e., where one level influences the elements in more than just the immediate level below it in the modeling hierarchy.

2 Example domain model

As a sample model we adopt the computer hardware product hierarchy described by Engels et al. [9]. This is a good example because Engels et al. clearly present a small and easy to understand example that accurately captures the state-of-the-art in modeling today. In particular, it contains a typical technique which is commonly used to express a multi-level domain classification in terms of the two-level UML modeling approach. The domain types and their relationships are shown in the class diagram of Fig. 1 and a possible

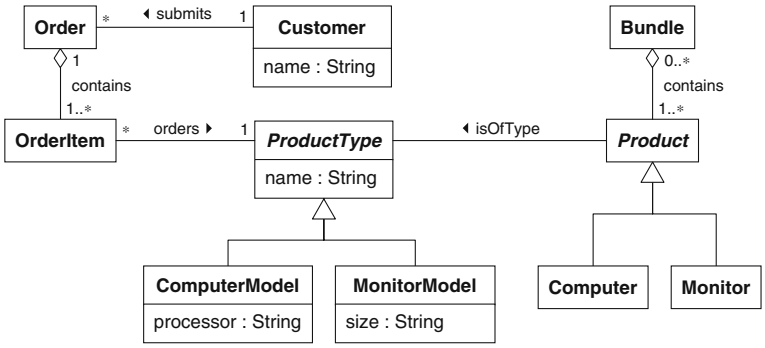

Fig. 1 Domain types (from [9] with name changes)

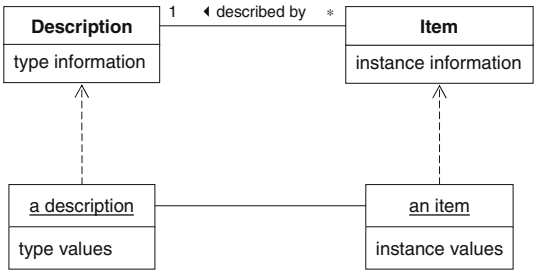

Fig. 2 Item description pattern

configuration of domain instances is shown in the object diagram of Fig. 3.[1]

The class diagram revolves around two hierarchies which are commonly found in static models of enterprise information systems—one modeling the products sold by the enterprise (physical objects, such as specific monitors and computers) and the other modeling the descriptions for these products (specifications of the physical objects, such as brand and model attributes). This is an example of the "Item Description" pattern [7] in which instances of the description types (e.g., instances of "ComputerModel" and "MonitorModel") are specifications of instances of the product types (e.g. instances of "Computer" and "Monitor"). Figure 2 shows the general structure of the "Item Description" pattern which is also known as the "Type Object" pattern [15].

The general idea is to let objects play the role of classes in order to explicitly represent class-level information, such as flight routes as opposed to actual flights [18]. In this way one can factor out information common to all objects to a single place, change it dynamically and keep this information even if no objects exist at a particular point in time. In our example, the (indirect) instances of class "ProductType" play the role of types for the (indirect) instances of class "Product" (see Fig. 3).

Several variations of this workaround pattern exist, testifying to the need to represent both instances and types of a domain. Together with the "Property Pattern" the "Type Object Pattern" [15] supports the creation of an "Adaptive Object-Model" and "User Defined Product" frameworks [31], i.e., the runtime creation of types with a dynamic specification of their "attributes". In such scenarios, the "Type Object" pattern is applied twice in a "type square" [31] to make type properties as dynamically flexible as types. The "Dynamic Object Model" architecture describes how "Type Object" may be combined with other patterns [29]. The "Dynamic Template Pattern" [20] addresses inheritance between represented types.

The literature describes a number of real usages [31] and example applications [15] of "Item Description" like solutions. Larman identifies a common need in "sales, product, and service domains" and manufacturing [18]. Frank also presents motivation, requirements, and consequences of a number of corresponding implementation strategies in the context of e-commerce applications [10].

Figure 3 shows a snapshot of a configuration of our example at the instance level. The instances of "MonitorModel" and "ComputerModel" define the characteristics of particular models of monitor and computer respectively. The instances of type "Monitor" and "Computer" represent products, each with a link to their respective *description* which indicates what kind of product it is and what shared information it is associated with. The "isOfType" links between "Monitor"/"Computer" instances and "MonitorModel"/"ComputerModel" instances thus conceptually represent a form of "instance of" relationship, even though all the objects involved are at the instance level in an object diagram.

3 Two-level modeling

The diagrams in Fig. 1 and 3 apply the UML in a widely accepted way, based on the underlying "two level" object-oriented modeling paradigm. In this section, we evaluate the strengths and weaknesses of this approach. In Sects. 3 and 4 we then discuss how the identified shortcomings can be addressed.

[1] We have used the exact models of [9] with the exception of some slight renaming of a few concepts.

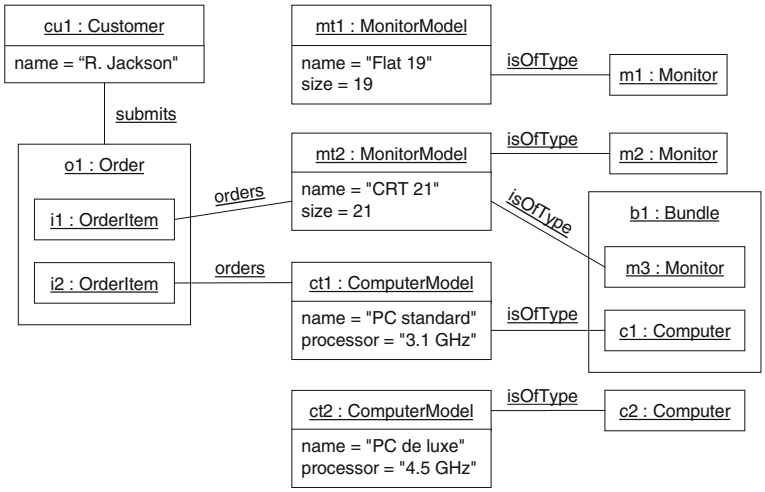

Fig. 3 Domain instances (from [9] with name changes)

3.1 Representing domain types as objects

The UML, being based on a fundamental distinction between classes and objects, supports a "built-in" notion of classification with two notations—one textual and one graphical—for expressing that an object is described by a class. In the textual form the object's type is specified after its name (see Fig. 3), and in the graphical form it is shown via a dependency relationship, stereotyped with "«instance of»", pointing from the instance to the type. In the following we will use the graphical form but will omit the stereotype since we use no other kinds of dependency relationships.

The "Item Description"/"Type Object" patterns represent an instance/type relationship using links between objects (see the "isOfType" links in Fig. 3) rather than using the "built-in" way of representing classification. These "isOfType" links are classified by the corresponding association between "Product" and "ProductType" (see Fig. 1). The purpose of this approach is to *model* the instance/type relationships at the object level. When considered together, therefore, the class diagram and object diagram in Figs. 1 and 3 embody three distinct kinds of "instance of" relationships:

1. the UML's "built-in" "instance of" relationships between elements in the object diagram and elements in the class diagram (e.g., "m1" is an instance of "Monitor").
2. the "isOfType" associations in the class diagram between Product classes and ProductType classes (e.g., "Monitor" "isOfType" "MonitorModel").
3. the "isOfType" links between instances of Product and instances of ProductType (e.g., "m1" "isOfType" "mt1").

Whenever a "built-in" concept (here, the UML's instantiation concept) is passed over in favor of some user defined replacement (here, the modeling of "instance of" relationships as associations and links), there is the possibility of undesired additional complexity creeping in. To clarify the effects of this approach, we combine the diagrams of Figs. 1 and 3 into a single diagram (see Fig. 4), so that *all* of the relationships that conceptually exist between the various model elements are explicitly represented. We also show the correspondence of the modeling elements to the domain entities they model. To save space we omit four classes from Fig. 1, focusing exclusively on products and product types. The omission of these classes does not affect the validity of the analysis.

Figure 4 highlights two aspects of the "two level" modeling paradigm which are not explicit in Figs. 1 and 3. First, it depicts all UML "instance of" relationships (as dashed arrows) in addition to the modeled "isOfType" relationships (the links and the association of that name). Second, it highlights the fact that the object diagram contains "model" information that represents domain instances *and* types of the domain conceptualization. In Fig. 4, these domain entities are shown as black dots below the big horizontal dashed line. We therefore view the corresponding model types as representations of domain types. This is why we also use the same "instance of" notation between domain entities (the dashed lines between the black dots). These relationships show which domain entities are considered to be types of other domain entities and thus represent *ontological "instance of" relationships* [3]. These are technically also known as relationships that do not cross language definition boundaries, i.e., are *intra-level instance-of relationships*, e.g., "snapshot" relationships within level M_1 of the OMG's four-layer architecture [16].

When the class and object diagrams are presented together as in Fig. 4, one striking consequence of using the "Item Description" pattern becomes evident: the resulting model features a considerable level of redundancy. First, there are model elements that do not represent any domain elements. The fact that the instances of "Product" and its

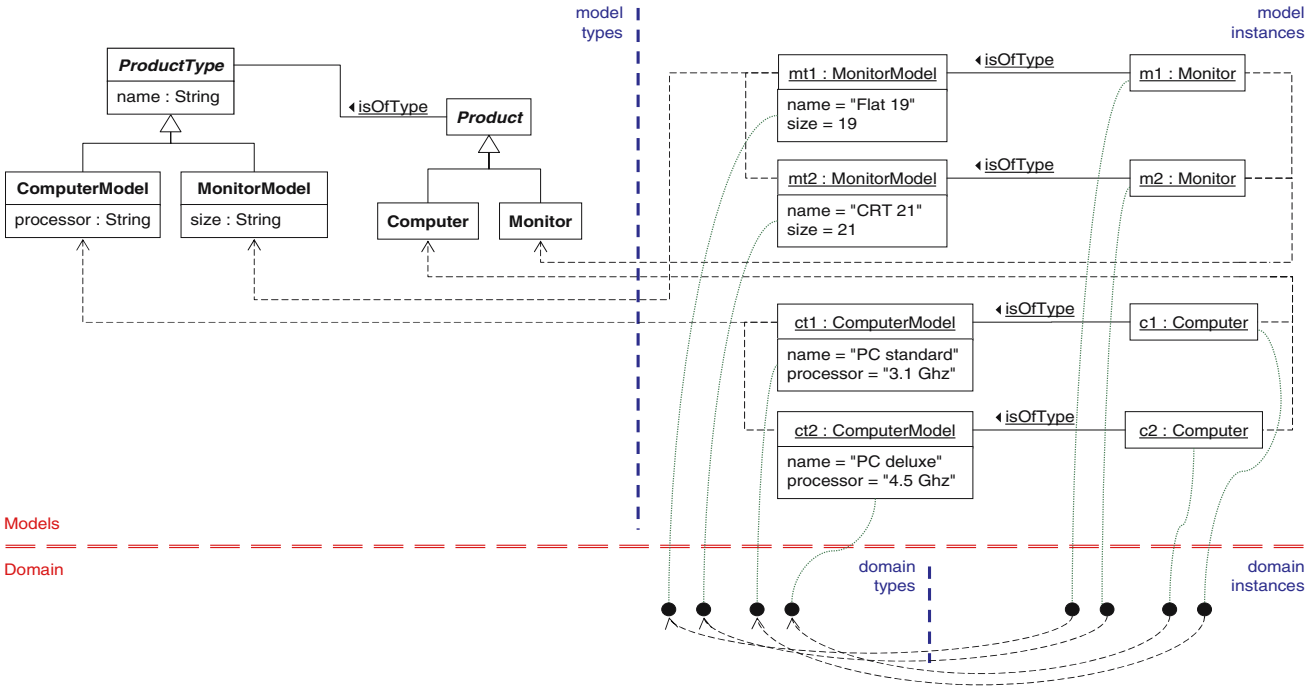

Fig. 4 Types as objects

subclasses have two classification relationships instead of just one indicates that "Product" and its subclasses are not representations of true domain entities but only serve to "set up" the "Item Description" pattern. For example, "m1" has type "Monitor" plus a modeled type "mt1". In terms of modeling the domain, only type "mt1" is of relevance. Type "Product" and its subclasses can hence be considered to be *abundant model attributes* [19], if the model is interpreted as a domain model, rather than a design for an object-oriented implementation. Second, the model not only contains more "instance of" relationships than the domain, it uses three different ways to represent them (UML instantiation, associations, and links). This can be considered to conceptually correspond to *construct redundancy* [30]. The model is therefore considerably more complex than its subject in the sense that it includes more elements and employs more concepts (in terms of different kinds of instance of relationships).

Such a situation is often an indicator of a mismatch between the structure of the problem and the technology used to represent it. To avoid this, the goal in software engineering—indeed in all engineering disciplines—is to find problem representation technologies that minimize the *non-inherent* complexity embedded in engineering systems, also called the *accidental complexity* [6].

In the context of this paper, we define the *accidental complexity* of a domain model as the amount of information in a model that is not induced by the corresponding domain conceptualization but, for instance, only exists to realize some workaround technique, the latter being used because an isomorphic relationship between domain concepts and modeling language concepts is not possible. The degree of accidental complexity thus correlates in an absolute manner with the number of elements and relationships in a model, and can be determined by comparing the latter with the number of domain elements and relationships to be represented.

In a relative sense, one domain model will consequently have a higher degree of accidental complexity than another if it features more modeling elements, provided both contain the same amount of information about the domain. Note, however, that comparing the number of elements of two models provides only an indirect measure of their respective accidental complexity. The latter depends on how directly and faithfully a domain model can capture the conceptualization it represents, in other words, on the relationship between a domain model and its subject, not on the domain model itself. This is the reason why purely counting model elements or the application of some other standard metric to a domain model may yield some coincidental correlation but will not directly measure domain model complexity.

3.2 Representing domain types as classes

Strategies similar to the "Item Description" pattern as employed in Figure 1 are often used in situations where new types need to be introduced dynamically. For example, one may want to create "MonitorModel" instances at runtime (e.g., "PlasmaPanel") to allow the creation of new monitor instances (e.g., "m3") of such new monitor types. This level

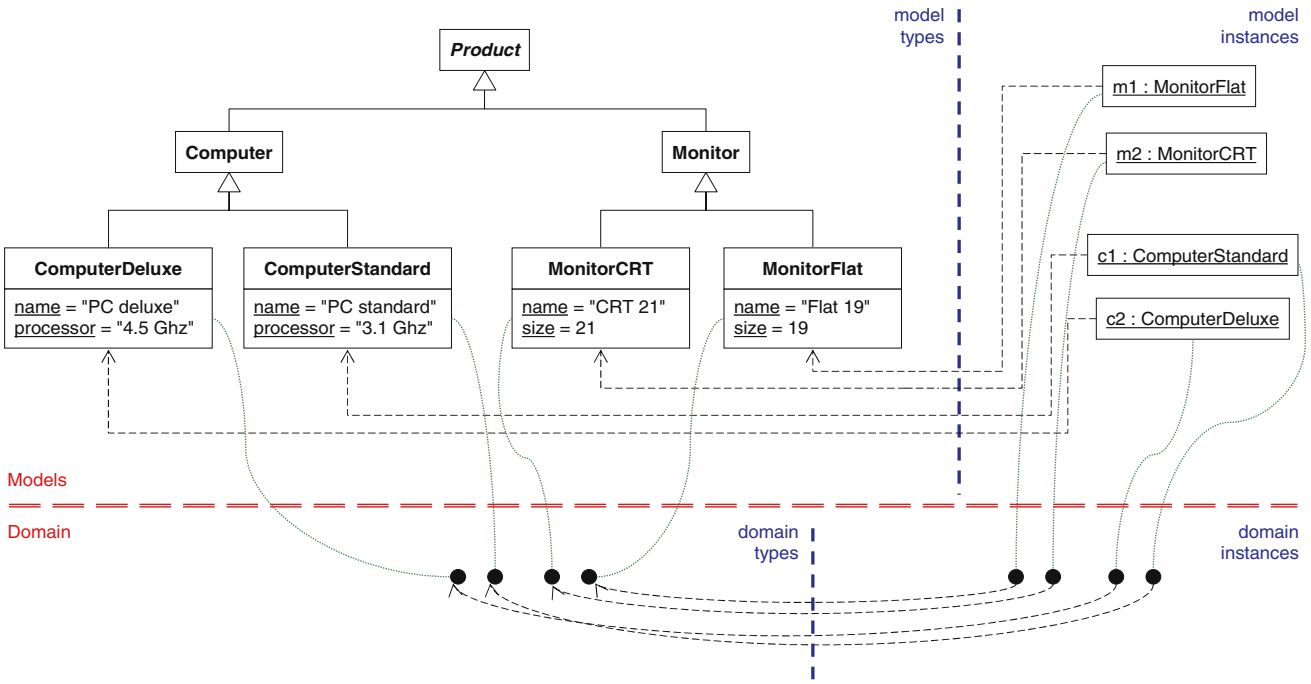

Fig. 5 Types as classes

of dynamic flexibility, however, is not needed in all cases. Sometimes it is sufficient to be able to specify in one place the values that are shared between all instances of a particular type. Whenever this is the case, as might reasonably be the case in our example, then a simpler modeling solution that does not employ "Item Description" or similar patterns is possible.

Figure 5 shows such a simpler approach, which lifts the modeling elements that represent domain types (e.g., "mt1") to the type level (i.e., into the class diagram), where they more naturally belong according to UML's class/object modeling conventions and built-in classification mechanisms. By modeling the elements representing different product types (such as "MonitorCRT" and "MonitorFlat") as classes we can again use UML instantiation between product types and instances (e.g., between "MonitorFlat" and "m2") and the whole model becomes significantly simpler.

Figure 5 introduces "Product", "Computer", and "Monitor" as superclasses for the respective product types. This compensates for the removal of "ComputerModel" and "MonitorModel" from the object diagram (see Fig. 4). The product types are thus no longer distinguished by being instances of "ComputerModel" or "MonitorModel" but by being subtypes of "Computer" or "Monitor".

Figure 5 certainly shows a simpler model than that of Fig. 4 in the sense that it contains fewer modeling elements and expresses all conceptually existing "instance of" relationships in a uniform way using UML's built-in classification relationship. According to the definition of complexity in the previous section, one may therefore consider the model of Fig. 5 to have less accidental complexity. However, it is only reasonable to compare the complexity of models when judging them against the same requirements. If there is no requirement that new product type instances be dynamically creatable, then Fig. 5 can be viewed as an equally accurate but more concise version of Fig. 4. This being so, it would then be reasonable to view the model of Fig. 5 as having lower accidental complexity. If, however, there is a requirement for the dynamic generation of product types then this interpretation would not be appropriate since the model shown in Fig. 5 does not provide the same capabilities as the model shown in Fig. 4.

The model of Fig. 5, if interpreted as a design model for a software system, describes a system without the capability of creating types at runtime, because of an underlying assumption that the type level (top left compartment of Fig. 5) is static, i.e., not extensible at runtime. The model of Fig. 4, again interpreted as a design model for software development, supports the dynamic creation of new domain types by representing these types explicitly as objects. There is no hard and fast rule that states that the type level is static, but this is the usual convention of modelers. With the requirement for dynamic instantiation of types, Fig. 4 must hence be regarded as a more accurate model of the scenario than Fig. 5, and it would thus be inappropriate to regard its extra complexity as "accidental". On the contrary, the model of Fig. 5 should not, in this case, be regarded as an accurate model.

It is indeed a shortcoming of the models as shown in Figs. 1 and 3 that we cannot know whether dynamic type creation should be supported or not. Without further documentation we cannot know which of the workaround properties are intended or accidental. This not only concerns dynamic type creation but also whether or not type properties (such as "size" for monitor models) are constant and/or can be accessed by their respective instances.

In Fig. 5, we have chosen to represent the values associated with product types in Fig. 1 as "static attributes", since this is the most concise way of defining changeable properties that are accessible by, but the same for, all instances of a class. An alternative, when using the UML, might have been to assign "default values" to regular class attributes. However, while this would have ensured that all instances obtained the corresponding value at their time of creation, it would allow each instance (such as "m1") to change the value individually. Defining the attributes to be constant would remove the latter problem but would also globally remove the ability to change the value at run-time. UML's tagged values offer a third way of associating properties with classes, being appropriate when properties are neither dynamically changeable nor accessible to instances of the class.

4 Three-level modeling

In Fig. 5, we were able to reduce the accidental complexity of the domain model by raising product types to the traditional class level and capturing the relationships between products and product types by using UML's built-in instance-of relationship rather than modeling some of them with associations/links. However, this could only be interpreted as a simplification of the model under the assumption that there was no requirement for dynamic generation of new product types.

The obvious question is whether it is possible to retain the simplicity gained in Fig. 5 by moving the product types to the traditional class level while simultaneously supporting the ability to create new product types dynamically. If we assume the type level is just as dynamic as the object level, we need some control over what types are created and what features they have. In other words, we need (domain-) metatypes controlling the features of (domain-) types. In fact, the introduction of metatypes will not only enable dynamic type creation but also reveal why the original design of Figure 1 displayed accidental complexity.

4.1 The level mismatch problem

The problem of solutions using strategies similar to the "Item Description" pattern is that they are attempting to capture a domain scenario inherently involving three classification levels using mechanisms that were designed to support a two level (class/object) paradigm. Figure 6 illustrates the mismatch between the number of levels schematically. It shows that the domain scenario of our example actually features *three* ontological levels—domain instances, their types, and the types of the types, i.e., metatypes—while the corresponding models actually only contain *two* levels (as defined by the fundamental class/object dichotomy of the UML). This means that somewhere *one* modeling level must be used to represent *two* domain levels. Figure 6 shows how the original design of Fig. 1 represents both domain types and instances at the modeling instance level, while domain metatypes are represented at the modeling type level.

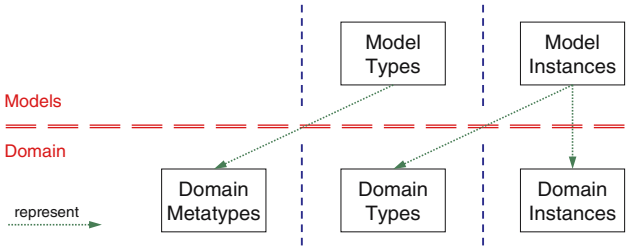

Fig. 6 Mapping domain levels to modeling levels

We can therefore observe that whenever there is a need to represent a domain with more than two inherent classification levels with a two level paradigm, some artificial "trick"/"workaround" has to be used to squeeze the three or more levels into two levels. The application of the "trick" invariably introduces some accidental complexity and typically makes it impossible to distinguish between the intended modeled properties and those that are not required but "trick"-induced.

In the following we look at two UML mechanisms—compared and linked to each other in [13]—to support more than two modeling levels and evaluate their suitability for capturing our example domain scenario.

4.2 Representing domain metatypes as powertypes

Powertypes as introduced by Odell [24]—and later incorporated into the UML—appear to be appropriate for addressing the representation of domain metatypes, which we need to control the dynamic type level. Just like a metatype, a powertype has instances that are types and may be further instantiated. The powertype mechanism is conceptually identical to the "materialization" mechanism [28], which has been independently developed in the database community [12]. Both approaches establish a relationship between a type (e.g., "MonitorModel") and a supertype (e.g., "Monitor"), where the latter's subtypes are regarded as instances of the former. In the following we focus on powertypes and in particular their

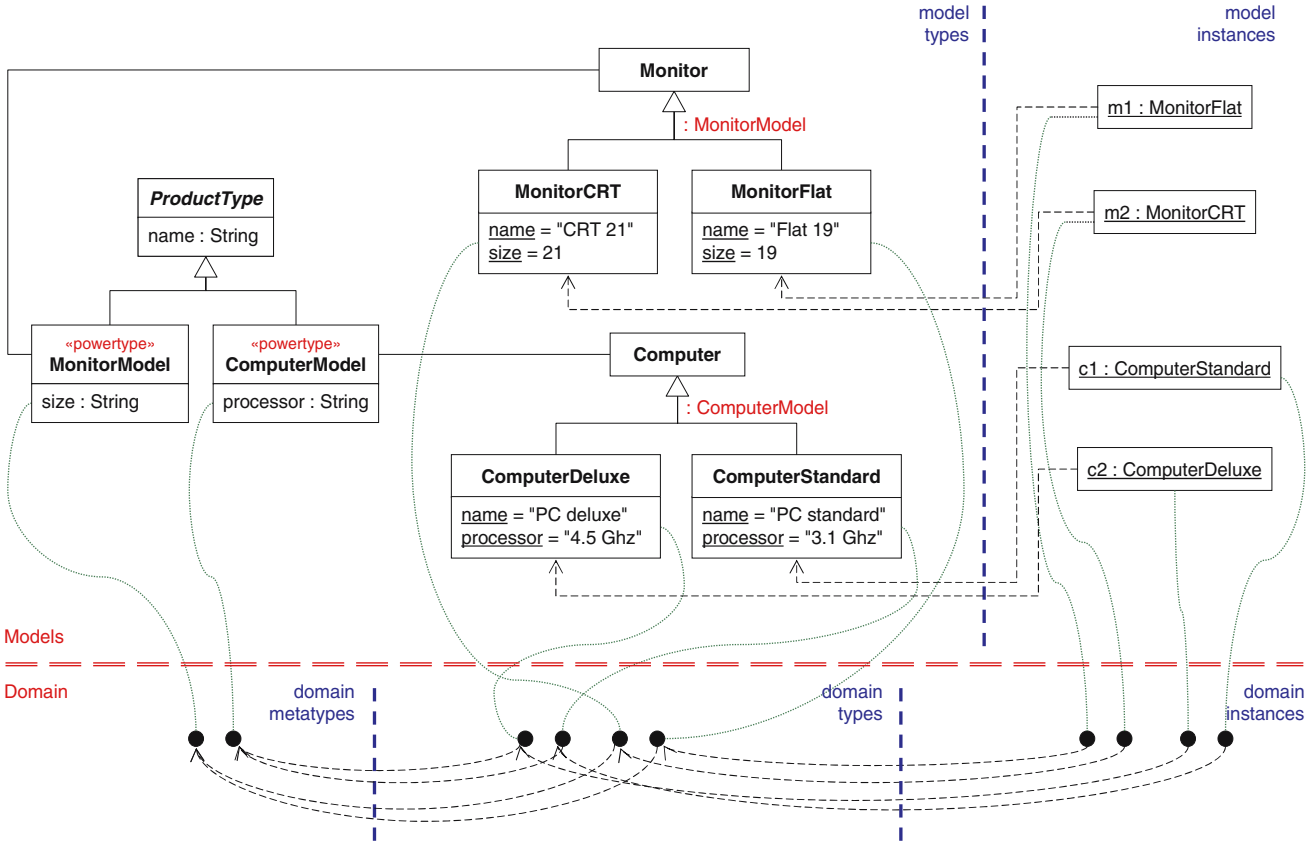

Fig. 7 Metatypes as powertypes

integration with the UML. However, the main arguments also apply to other incarnations of the approach, including the "materialization" mechanism.

Although the model in Fig. 7 superficially fulfills the goal of defining (meta-)types for the subclasses of "Computer" and the subclasses of "Monitor", closer examination reveals some problems. The first problem of powertypes as adopted in UML 2 is the use of an additional mechanism to describe the "instance of" relationship. This not only entails construct redundancy [30], but in this case is a less than satisfactory substitute for the standard class/object relationship for the following reason: Since the relationships between "Computer" & "ComputerModel" and "Monitor" & "MonitorModel", which are intended to express classification, are modeled in the form of (powertype-) associations rather than in the form of the standard class/object relationship, the attributes of metatypes (ProductType and its subtypes) cannot specify properties in their corresponding instances (e.g., "MonitorCRT" or "ComputerDeluxe") according to regular UML semantics. Although the UML specification claims that powertypes may introduce class properties (as known from [24]), it is unspecified how this is supposed to be supported by concrete UML semantics. Officially, therefore, one cannot assume that UML powertype attributes are able to control type-level properties of their instances. This may make UML powertypes appear useless, but in section 5 we will actually see how they can be of real value.

The second problem of powertypes as supported by UML 2 again relates to the use of associations to denote the "instance of" relationship between a powertype and the superclass of its instances. Despite the further occurrence of powertypes names as generalization discriminators (see Fig. 7) this achieves too little from a semiotic perspective to signify the fundamental difference between a powertype and its instances. As a result, both notationwise and semantically (because of problem one), powertypes, as interpreted by UML 2, appear to be closer to supporting a two-level rather than a three level approach.

The second problem associated with the design of Fig. 7 stems from the *mandatory* use of superclasses "Computer" and "Monitor". If the latter exist in the conceptualization of the domain since they are thought to be useful, then their presence is unproblematic. If, however, the conceptualization does not contain them, their introduction in order to support the powertype mechanism clearly amounts to adding accidental complexity.

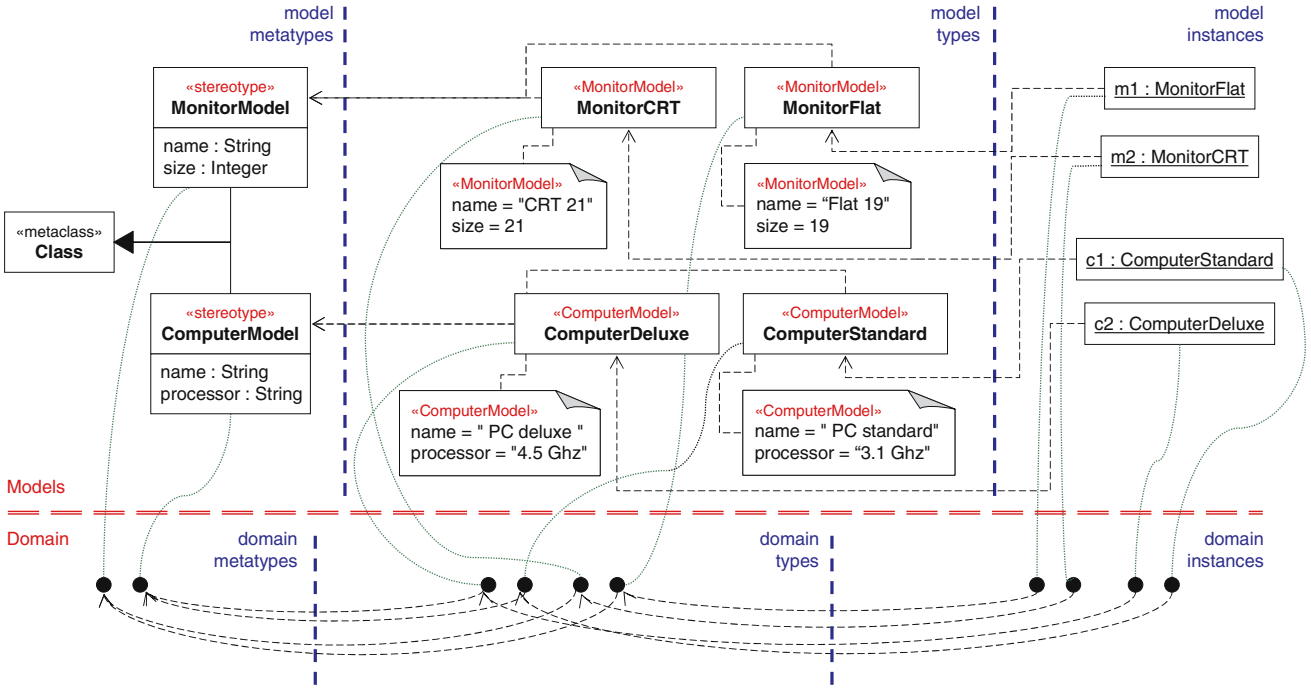

Fig. 8 Metatypes as stereotypes

4.3 Representing domain metatypes as stereotypes

Strictly speaking, the intention of stereotypes is to support language extension. Therefore, instead of providing support for representing ontological domain metatypes, technically they extend the abstract syntax of the language by introducing new linguistic (meta-)types [3]. Nevertheless, stereotypes may be (and in practice *are*) used as a "poor man's" way to support domain metatypes. Figure 8 shows how stereotypes can be used to "brand" regular classes with metatype names and to equip them with "tagged values".

Although the effect of stereotypes—when used as in Fig. 8—may be *conceptually* regarded as adding a third user level, they do little to reduce the accidental complexity of domain models. There are two reasons why. First, they involve the use of a new concept for classifying modeling elements with subtle differences from the standard one, introducing yet another way of representing the instance-of relationship (since a stereotype can be considered to classify the stereotyped element [5]). Again, this is a form of construct redundancy [30]. Second, their most direct way of associating properties with classes, the so called tagged value mechanism, is quite limited. As discussed at the end of Sect. 3.2, tagged values only support immutable class properties that cannot be accessed by instances of the respective classes. Of course, stereotypes may also be combined with OCL statements, which could be used to enforce other ways of supporting class properties, but that would only address the accuracy of the model, not positively influence its level of accidental complexity.

5 True multi-level modeling

The patterns and mechanisms discussed in the previous two sections represent the state of practice today. However, they tend to increase the accidental complexity of domain models by (a) adding "verbosity" due to an increased number of modeling elements and concepts needed to represent a domain, and (b) by obscuring the domain information in a model, making it a less accurate representation of the domain and/or unfit for specifying what properties are actually required from a solution.

The root of the problem is the use of various "artificial" ways of representing the "instance of" relationships within a model, and the dilution of the fundamental class/instance level boundary which this entails. This is in turn a symptom of the lack of support—within the core modeling concepts of languages like the UML—for representing multiple classification levels within a domain. There are of course knowledge representation languages (e.g., Telos [23]) and supporting systems (e.g., ConceptBase [14]), that do not suffer from such a two-level limitation, but these are not used in practice in the context of software engineering because they are not known and/or not optimized for such applications.

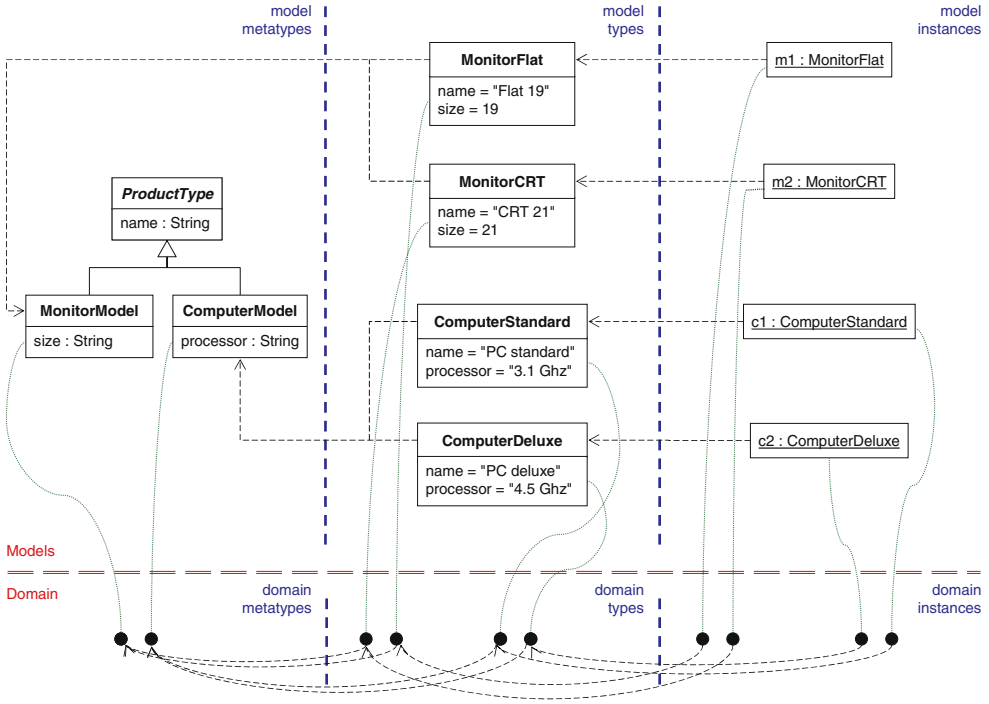

Fig. 9 Uniform multi-level support

In the following we show how uniformly supporting more than two modeling levels within a modeling language allows the accidental complexity of domain models to be minimized.

5.1 Uniform multi-level support

In order to truly move beyond the traditional "two level" paradigm, it is necessary to provide modeling concepts that can be applied in a uniform way across all levels in a multi-level classification hierarchy. To do this, it is necessary to have a modeling construct that supports the representation of the dual "type and object"-property of some domain concept. In [1], we refer to such constructs as clabjects (*clas*s and o*bject*) and represent them using a combination of notational conventions from UML classes and objects. Like classes, clabjects have a name and a set of attributes, listed in an underlying compartment. Like objects, clabjects can have a set of slots, listed above the attributes.

Figure 9 illustrates the example we have been studying, modeled in terms of clabjects. We now no longer need to resort to the UML concept of static class attributes since class properties are naturally supported by clabjects, simply as class-level slots. Note that because the instances of the original example specified in Fig. 3 (see the rightmost column of objects) do not have any slots, the clabjects in the middle column of Fig. 9 do not need to have any type facet, i.e., they do not specify any attributes.

By representing all domain concepts with clabjects and using a single notion of "instance of" relationship, the model of Fig. 8 features less accidental complexity than the solutions presented in Sects. 3 and 4. Since clabjects naturally support any number of model classification levels, the latter can be directly aligned with the domain classification levels. Figure 10 highlights the corresponding *direct mapping* between ontological domain levels and modeling levels.

The principle of "direct mapping" as defined by [21] to characterize software modularity requires that the solution structure reflect the problem structure as closely as possible and leads to solutions that are easier to understand and are modifiable in a more localized manner. This principle, hence, nicely matches the notion of accidental complexity in that one must maximize the former to minimize the latter.

The models of Figs. 7 and 8 approach this "direct mapping" quality but do not fully achieve it due to the half-hearted

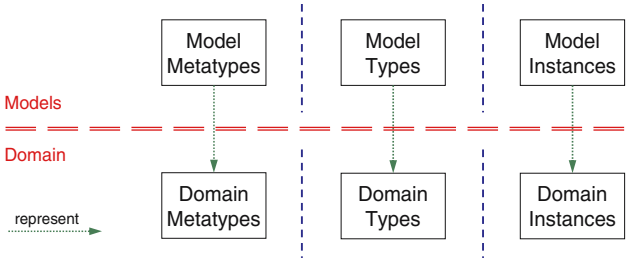

Fig. 10 Direct mapping of domain levels to modeling levels

integration of powertypes with their insufficient separation from types and the ad-hoc notion of stereotypes which are really defined at the user type level respectively. Only the effect of introducing stereotype definitions, i.e., a virtual extension of the UML metamodel, can be displayed as shown in Fig. 8.

Now that we have identified a clean way of supporting multiple domain levels, we can move on to discuss support for "deep characterization", i.e., the phenomenon occurring whenever elements in one level need to influence the characteristics of elements beyond those in the level immediately below.

5.2 Deep characterization with powertypes

The modeling approach used in Fig. 9 captures all the information conveyed in the original model of Figure 4 in a more direct and concise way. We therefore believe it already represents a significant step forward over the current practice in domain modeling. However, there is one additional issue which needs to be addressed in order to allow all aspects of domain scenarios involving multiple classification levels to be fully and naturally modeled. This is the issue of *deep characterization* [17].

Deep characterization exists when a type in a scenario with multiple classification levels wishes to influence (i.e., make statements about) entities beyond its immediate instances. Technically, deep characterization means that a type can influence its instances' type facets (e.g., the attributes of types, controlling the shape of further instances) as well as their object facets (i.e., properties with values).

In order to demonstrate deep characterization we extended our example with "price" and "picture quality" ("picQlty") attributes for classes "Computer" and "Monitor" classes respectively. This assumes that particular monitor instances are distinguished by the individual picture quality they offer, because they may have individual geometry and/or "dead pixel" problems. In analogy, particular computer instances may be assigned individual prices because of their condition/appearance. This is a case of deep characterization since we want to associate with "ComputerModel" and "MonitorModel" the fact that any of its instances has to define attributes "price" and "picQlty" respectively. In other words, that all instances of "ComputerModel" instances will have a "price" property and all instances of "Monitor" instances will have a "picQlty" property.

Without explicit language support to address this requirement, the best way to model deep characterization today is to use the powertype mechanism as discussed in Sect. 4.3. Figure 11 shows how the powertype mechanism can be used to represent deep characterization. If a superclass (e.g., "Computer") is said to have a powertype (here "ComputerModel") then an instance of the powertype (e.g., "ComputerStandard") is not well-formed unless it inherits from the superclass ("Computer"). In our example, every subclass of "Computer" and "Monitor" must also be an instance of powertypes "ComputerModel" and "MonitorModel" respectively. Hence, all subclasses will have a "processor" or "size" *slot* and a "price" or "picQlty" *attribute*. Powertypes therefore control the type facet of powertype instances by means of inheritance. In combination, powertypes and supertypes fully define the powertype instances regarding their instance facets and type facets respectively.

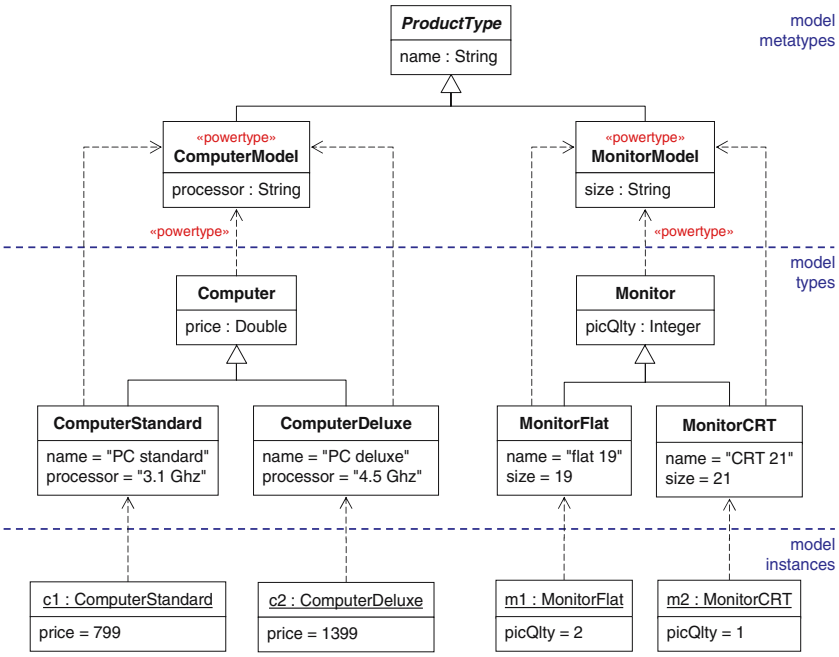

Fig. 11 Deep characterization with powertypes

Note that we have given powertypes the benefit of a clearer presentation by using an outdated UML 1.1 notation (a dependency relationship stereotyped with "«powertype»") [25] and placing them at an ontological user metatype level. A UML 2 conformant model (such as the one in Figure 7) has to use associations between powertypes and supertypes and effectively places both at the same user level [27].

5.3 Deep Characterization with Deep Instantiation

The powertype mechanism supports deep characterization but at the potential cost of introducing supertypes whose only purpose might be to provide a type facet for their subclasses. As already argued at the end of Sect. 5.2, these supertypes may exist in the domain conceptualization for a good reason in any case but, if not, they add accidental complexity to the domain model since they are only being introduced because of the idiosyncrasies of a particular solution to deep characterization. Moreover, the effective application of the powertype mechanism requires modelers to understand the rules of this mechanism, which go beyond basic instantiation rules.

To represent deep characterization with guaranteed minimal accidental complexity, we need a mechanism that supports it in as direct and concise a way as possible. We have presented such a mechanism in earlier work [2] and refer to it as *deep instantiation*. Deep instantiation essentially embodies two ideas—one is the unification of (meta-) attributes and slots into a single, more general concept, which we refer to as "field"; the other is the idea of assigning an additional property to clabjects and fields known as "potency", which defines how deep the instantiation chain produced by that clabject or field may become. For example, a field of potency two can produce an instantiation depth of two. The first time it is instantiated, the potency-two field causes the creation of a potency-one field of the same name (yielding a regular attribute). When the latter is instantiated again, it causes the creation of a potency-zero field (yielding a regular slot). Entities with potency zero cannot be further instantiated just like regular objects or slots. Further details about *deep instantiation*, including a formal semantics, may be found in [2,17].

Figure 12 shows how deep instantiation can be used to represent deep characterization in our example. Clabject "MonitorModel" has a field "picQlty" of potency two, indicated by the superscript "2" at the field name. When "MonitorModel" is instantiated to create "MonitorFlat", all its fields with a potency higher than zero are copied to "MonitorFlat", with their potencies reduced by one. Thus "picQlty" becomes a field of potency one (i.e., the desired attribute at this level) while "size" becomes a field of potency zero (i.e., a slot). The further instantiation of "MonitorFlat" to generate "m1" reduces the potency of "picQlty" to zero and turns it into a regular slot.

Note that superscripts with the values "1" and "0" may be omitted where the context makes the choice clear, for example for slots of potency zero within an object of potency zero. In our example, we therefore only have to specify the potency-two elements and, additionally, demonstrate the use of a potency value of zero for clabjects with non-zero potency fields, by specifying "ProductType" in Fig. 12 to be an *abstract* (meta-) class.

The version of the domain model depicted in Fig. 12 is not only the model with the least accidental complexity of all the versions we have discussed, it also is more expressive than the original design (see Fig. 4) because it features deep characterization. It furthermore only allows computer and monitor types for computer and monitor instances respectively. In comparison, the model of Fig. 4 needs to be enhanced

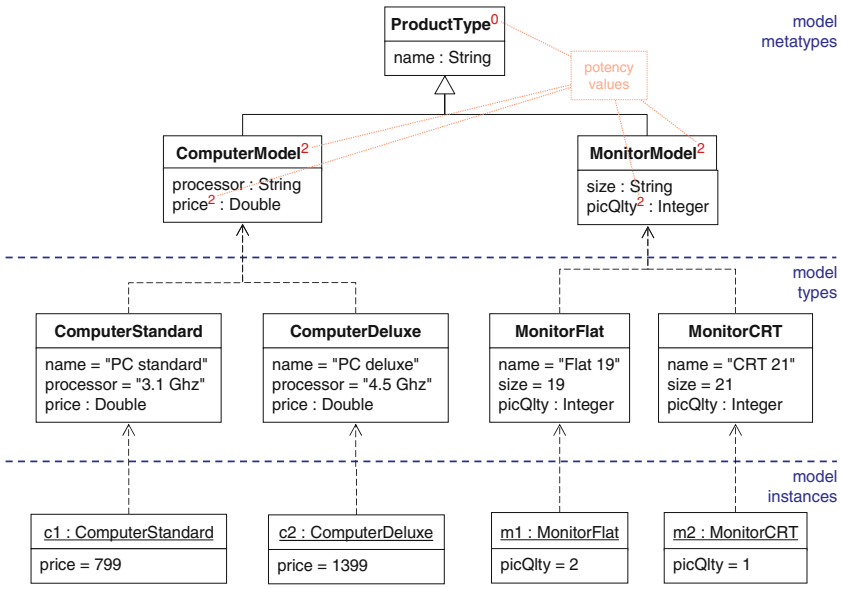

Fig. 12 Deep characterization with potency

with further constraints, or concepts like association inheritance, to prevent monitor instances being assigned a computer type by mistake. If deemed useful, concepts "Computer" and "Monitor" may be added to the model of Fig. 12 but one is not forced to do so. With a powertype-based solution, however, one is forced to add them. Either way, the description of entities such as "Computer" may be concentrated at a single place ("ComputerModel") when using deep instantiation. Powertypes, in comparison, require splitting the description of instance and type facets to elements "ComputerModel" and "Computer" respectively. Gonzalez-Perez and Henderson-Sellers, on the other hand, argue that the "splitting"-property of powertypes are a feature, not a bug [11]. However, if one accepts the principle that adding additional model elements to a model simply to "set up" the use of a particular technology (rather than because they naturally exist in the domain of study) increases the accidental complexity of a model, then powertypes are clearly not as adequate as clabjects and potency for the purpose of modeling problem domains.

6 Conclusion

In this article, we have shown that current modeling practices and many of the mechanisms supported by the UML are still rooted in the original "two level" paradigm which dominated the early modeling techniques, and is still the primary implementation model for mainstream object-oriented languages and databases. When the goal of a model is to describe a design targeted to such an implementation technology, this assumption of a basic two-level type/instance dichotomy is appropriate. In today's model-driven development terminology, such a model would be regarded as a platform-specific model, targeted to a "two-level" platform. However, when the goal of a model is to describe a domain scenario in a solution-independent way, any assumptions about specific implementation platforms are inappropriate because they introduce complexity that is not inherent in the domain—so-called accidental complexity.

We have defined the accidental complexity of a domain model as the extent to which it is capable of representing a domain conceptualization in the most direct way without adding any artificial, representation induced elements. By assuming a given conceptualization, whose quality we are not concerned with, we are thus measuring the *representation* quality of a model as opposed to the *content* quality of a model. As a result, standard software metrics are not directly relevant.

We believe the article's main contribution is in drawing attention to the "level mismatch" problem and the role that it plays in increasing the accidental complexity of domain models. This problem occurs whenever domain scenarios with three or more inherent classification levels are modeled in terms of mechanisms or concepts that are based on a "two level" modeling paradigm. To accomplish this, it is necessary to adopt one of several "workaround" techniques that allow one or more of the domain classification levels to be "folded" or "squeezed" into one modeling level. This not only obscures the underlying properties of the domain, it also leads to indirect mappings that are more difficult to maintain and check. In particular, such models are ill-suited to document the intent of the modeler with respect to the required solution properties. The reader can never be sure which of the model properties reflect the modeler's intent or are due to the idiosyncrasies of the "workaround".

We therefore proposed the support of a genuine multi-level way of modeling domain information which replaces the existing workarounds for "instance of" relationships explained in this paper with a single, fundamental notation that is uniform across all logical levels. Modeling at the level of metatypes is then no different to modeling at any other (type or instance) level and does not require any special modeling concepts, such as stereotypes or powertypes, thereby avoiding unnecessary *construct redundancy* [30]. The so-called *orthogonal classification architecture* [4] could be used as an underpinning modeling infrastructure, since it explicitly acknowledges the existence of multiple ontological domain levels and supports them directly with a uniform notation for all the levels and with a minimum of modeling concepts.

In Sect. 5, we demonstrated how—with the use of clabjects and potency—it is possible to create a model of the example domain with the least amount of accidental complexity, i.e., with the most direct mapping to the domain conceptualization it represents, using a minimal number of modeling elements. On top of this, the model was more expressive than all the other models that did not support deep characterization.

Note, however, that our main point is not to promote a particular technology, such as clabjects and potency, but to point out that there is a strong demand for some kind of technology supporting ontological multi-level modeling with concise support for deep characterization. Traditional "two level"-based approaches with their associated workaround mechanisms are simply a suboptimal choice for capturing the properties of a domain scenario in the most accurate way and for creating solution-independent models with a minimal amount of accidental complexity. In short, they are not suitable for fulfilling the underlying goal of domain modeling.

Certainly, capable approaches [23] and tools [14] addressing the above mentioned shortcomings exist, but the software modeling community has not taken advantage of them yet. We believe there are three main reasons for this. First none of the existing multi-level modeling approaches target software development as such. As a result the tools are not directly suitable for mainstream software development and—partly because of that—the software modeling community

is not generally aware of them. Second, the MDA research community has traditionally focused on metamodeling as a means for language definition rather than as an inherit part of end-user domain models. Third, the general software modeling community is simply not aware of the fact that "two level" approaches fundamentally impede the creation of optimal domain models.

Even if one intends to map the domain model to a solution technology that supports less levels than nominally required, we suggest that it is better to first create a clean model using as many levels as required and then—in a subsequent step—to transform this into a form that uses some of the available workaround practices where needed. The casting of a multi-level problem scenario into a two-level solution model would thus be regarded as part of the refinement process, not as something to be captured in the domain model itself.

References

1. Atkinson, C., Kühne, T.: Meta-level Independent Modeling, International Workshop "Model Engineering" (in conjunction with ECOOP'2000), Cannes, France (2000)
2. Atkinson, C., Kühne, T.: The essence of multilevel metamodeling. In: Proceedings of the 4th International Conference on the Unified Modeling Language, Toronto, Canada (2001)
3. Atkinson, C., Kühne, T.: Model-Driven Development: A Metamodeling Foundation. IEEE Softw. **20**(5), 36–41 (2003)
4. Atkinson, C., Kühne, T.: Concepts for Comparing Modeling Tool Architectures, In: Proceedings of the ACM/IEEE 8th International Conference on Model Driven Engineering Languages and Systems, MoDELS/UML 2005
5. Atkinson, C., Kühne, T., Henderson-Sellers, B.: Systematic stereotype usage. J. Softw. Syst. Model. **2**(3), 153–163 (2003)
6. Brooks, F.P.: No silver bullet: essence and accidents of software engineering. Comput. Archive **20**(4), 10–19 (1987) ISSN:0018-9162
7. Coad, P.: Object-oriented patterns. Commun. ACM **35**(9), 152–159 (1992)
8. Coleman, D., Arnold, P., Bodo, S. Dollin, C., Gilchrist, H., Hayes, F., Jeremaes, P.: Object-Oriented Development: The Fusion Method. Prentice-Hall, englewood Cliffs (1994)
9. Engels, G., Förster, A., Heckel, R., and Thöne, S.: Process modeling using UML. In: Process-Aware Information Systems. pp. 85–117 Wiley, Chichester,(2005)
10. Frank, U.: Modeling products for versatile e-commerce platforms—essential requirements and generic design alternatives. In: Arisawa, H., Kambayashi, Y., Kumar, V., Mayr, H.C., Hunt, I. (eds.) Conceptual Modeling for New Information System Technologies, pp. 444–456. Springer, Heidelberg (2002)
11. Gonzalez-Perez, C., Henderson-Sellers, B.: A powertype-based metamodelling framework. Softw. Syste. Model. **5**(1), (2006)
12. Goldstein, R.C., Storey, V.C.: Materialization. IEEE Trans. Knowledge Data Eng. **6**(5), 835–842 (1994)
13. Henderson-Sellers, B., Gonzalez-Perez, C.: Connecting powertypes and stereotypes. J. Object Technol. **4**(7), (2005)
14. Jarke, M., Gallersdörfer, R., Jeusfeld, M.A., Staudt, M., Eherer, S.: ConceptBase — a deductive object base for metadata management. J. Intell. Information Syst. Special Issue Adv. Deductive Object-Oriented Databases **4**(2), 167–192 (1995)
15. Johnson, R., Woolf, B.: Type Object, In Pattern Languages of Program Design 3, pp. 47–66. Addison-Wesley, Reading (1997)
16. Kühne, T.: Matters of (meta-) modeling. J. Softw. Syst. Model. **5**(4), (2006)
17. Kühne, T., Steimann, F.: Tiefe Charakterisierung. In Proceedings of "Modellierung 2004". LNI Vol. 45, pp. 121–133
18. Larman, C.: Applying UML and patterns: an introduction to object-oriented analysis and design and the unified process, 2nd (edn.) Prentice-Hall, Englewood cliffs (2002)
19. Ludewig, J.: Models in software engineering—an introduction. J. Softw. Syst. Model. **2**(1), 5–14 (2005)
20. Lyardet, F.: The dynamic template pattern. In: Proceedings of the Conference on Pattern Languages of Design (1997)
21. Meyer, B.: Object-Oriented Software Construction. Prentice-Hall, Englewood Cliffs (1997) ISBN 0-13-629155-4
22. Mittelstraß, J. (Ed.) Enzyklopädie Philosophie und Wissenschaftstheorie. Metzler Verlag, (2004) ISBN: 3476020126
23. Mylopoulos, J., Borgida, A., Jarke, M., Koubarakis, M.: Telos — a language for representing knowledge about information systems. In ACM Trans. Informat. Syst. **8**(4), 325–362 (1990)
24. Odell, J.: Power Types. J. Object-Oriented Program. (1994)
25. OMG: Unified Modeling Language, v1.1. OMG document ad/97-08-04, (1997)
26. OMG: MDA Guide Version 1.0.1, OMG document omg/03-06-01 (2003)
27. OMG: Unified Modeling Language, v2.0. OMG document formal/05-07-04, (2005)
28. Pirotte, A., Zimányi, E., Massart, D., Yakusheva, T:. Materialization: a powerful and ubiquitous abstraction pattern. In: Proceedings of the Conference on Very Large Database, pp. 630–641 (1994)
29. Riehle, D., Tilman, M., Johnson, R.: Dynamic object model, In: Pattern Languages of Program Design 5. Addison-Wesley, Reading (2005)
30. Wand, Y., Weber, R.: On the ontological expressiveness of information systems analysis and design grammars. J. Informat. Syst. **3**(3), 217–237 (1993)
31. Yoder, J. W., Johnson, R.: The adaptive object model architectural style. In: Proceeding of The Working IEEE/IFIP Conference on Softw. Architecture 2002 (WICSA3 '02)

Author's biography

Colin Atkinson heads the chair of Software Engineering at the University of Mannheim in Germany. Prior to that he was an Associate Professor at the University of Kaiserslautern and project leader at the affiliated Fraunhofer Institute for Experimental Software Engineering. From 1991 until 1997 he was an Assistant Professor of Software Engineering at the University of Houston - Clear Lake. His research interests are focused on the use of model-driven and component based approaches in the development of dependable computing systems. He received a Ph.D. and M.Sc. in computer science from Imperial College, London, in 1990 and 1985 respectively, and his B.Sc. in Mathematical Physics from the University of Nottingham in 1983.

Thomas Kühne is an Assistant Professor at the Darmstadt University of Technology in Germany. Prior to that he was an Acting Professor at the University of Mannheim (Germany) and a Lecturer at Staffordshire University (UK). His interests are centered on object technology, programming language design, model-driven development, component architectures, and metamodeling. He received a Ph.D. and M.Sc. from the Darmstadt University of Technology, Germany in 1998 and 1992 respectively.