Software and Systems Modeling (2022) 21:517–542
https://doi.org/10.1007/s10270-021-00938-2

THEME SECTION PAPER

# Accidental complexity in multilevel modeling revisited

**Mira Balaban**[1] · **Igal Khitron**[1] · **Azzam Maraee**[1,2]

Received: 26 May 2020 / Revised: 13 June 2021 / Accepted: 29 September 2021 / Published online: 18 January 2022


**Abstract**
Multilevel modeling (MLM) conceptualizes software models as layered architectures of sub-models that are inter-related by the instance-of relation. Conceptually, MLM provides benefits in terms of ontological classification. Pragmatically, based on arguments in knowledge engineering, MLM meaningfully reduces accidental complexity. In this paper, the problem of accidental complexity in MLM is revisited. The paper focuses on the role of the context of *type-instance* structures on MLM architectures. We analyze factors of accidental complexity in multilevel models, suggest quantitative metrics for these factors, and show how they can be used for guiding MLM rearchitecture transformations. The relevance of the proposed factors and metrics is shown in an experimental study of *type-instance* contexts in multiple real-world models.

**Keywords** Multilevel modeling · Context · Rearchitecture · Accidental complexity · Quantitative measures · Evaluation criteria

## 1 Introduction

Multilevel modeling (*MLM*) is a software modeling school that advocates multiple levels of ontological classification, similar to typical conceptualizations in natural domains. The intent is to enable domain specific *meta-types* that describe domain types [1]. Along with the philosophical modeling arguments, there came pragmatic engineering arguments that point to reduced *accidental complexity* [2] when two level models of *type-instance* relationships are rearchitected using multiple models [3–7].

Figure 1 shows an example, taken from [4] of two alternative models of a *type-instance* problem, and an alternative three-level model, following the *potency*-based approach (also termed *deep instantiation*) [8,9].

Communicated by Adrian Rutle and Manuel Wimmer.

This work is supported in part by BSF Grant 2017742.

✉ Azzam Maraee
mari@cs.bgu.ac.il

Mira Balaban
https://www.cs.bgu.ac.il/~mira

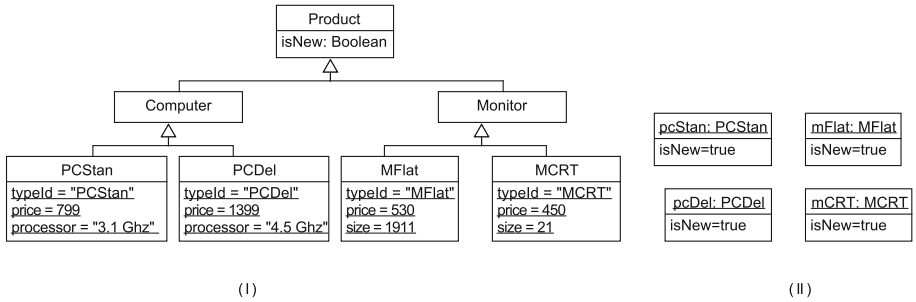
Igal Khitron
https://www.cs.bgu.ac.il/~khitron/

[1] Ben-Gurion University of the Negev, Be'er Sheva, Israel

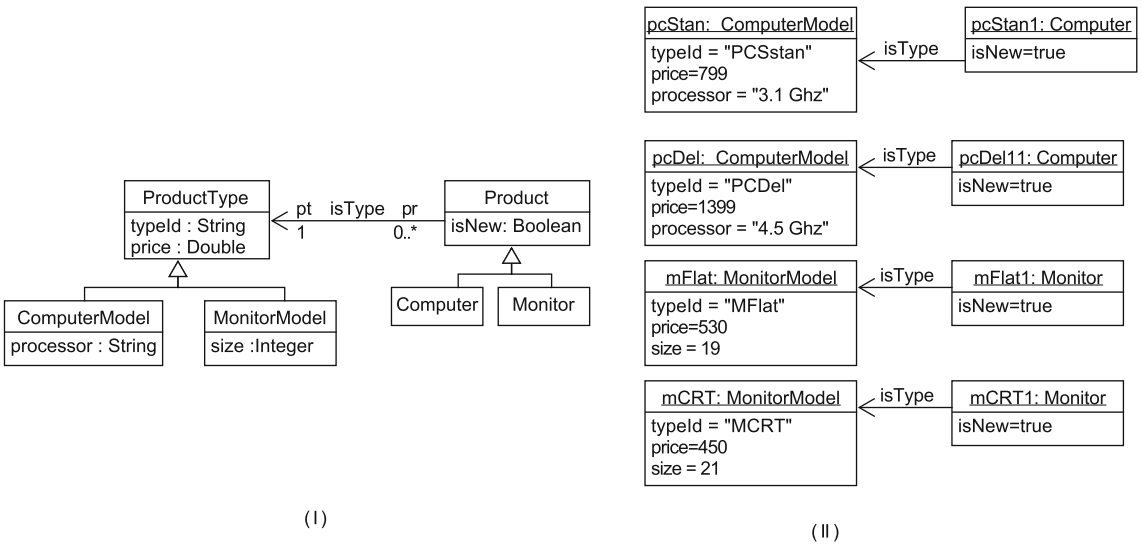
[2] Achva Academic College, Kiryat Malakhi, Arugot, Israel

The multilevel model consists of two class models, at Levels 2 and 1 (marked on the left corner as `L2, L1`). The diagram also shows an instance model at Level 0 (marked on the left corner as `L0`). Within levels, classes might be related by the *subtyping* relation, and between levels they are related by the *instance-of* relation, e.g., class *PCStan* on Level 1, which is an instance of class *ComputerModel* on Level 2. In general, classes and associations can be successively instantiated (and attributes can be successively inherited) down their level, but a *potency mechanism*, marked by "@n" sign, can restrict the number of successive instantiations of classes and associations, and control inheritance of attributes. For example, attributes *typeId* of *Producttype* and *processor* of *ComputerModel* are marked with potency 1, meaning that they are static on Level 1. The default potency of an element is its level and is not marked.

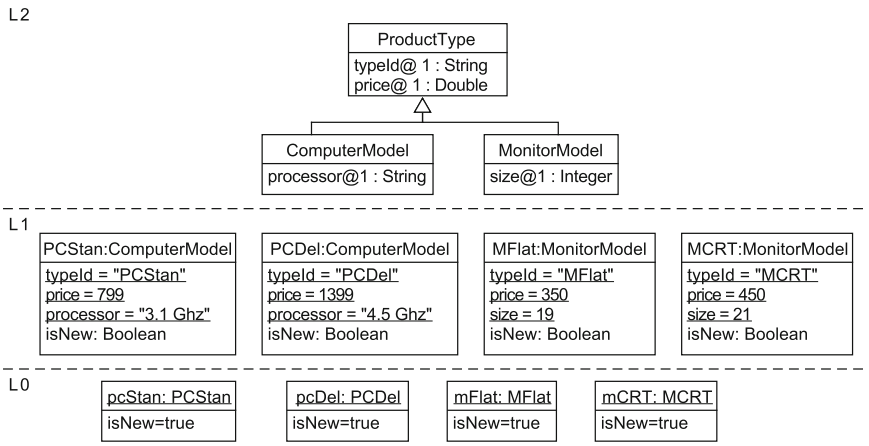
An MLM representation consists of a sequence (or sequences) of modeling levels. Each level (but the lowest) is used for modeling its lower levels, or alternatively, elements in each level (but the highest) function as instances of elements in higher levels. Overall, the models in a multilevel representation are inter-related by the *instance-of* relation among objects and classes, and maybe further restricted by inter-level and intra-level constraints. The exact relationship between adjacent levels varies among approaches. Some approaches do not require any prescribed relationship beyond the requirement that elements in level $i$ might be instances of elements in a level higher than $i$ [10]. Other approaches




**(a)** Class-hierarchy based model and a state




**(b)** Type-instance based model and a state




**(c)** Multilevel model

**Fig. 1** Multilevel rearchitecting of a class-hierarchy- and a type-instance-based models

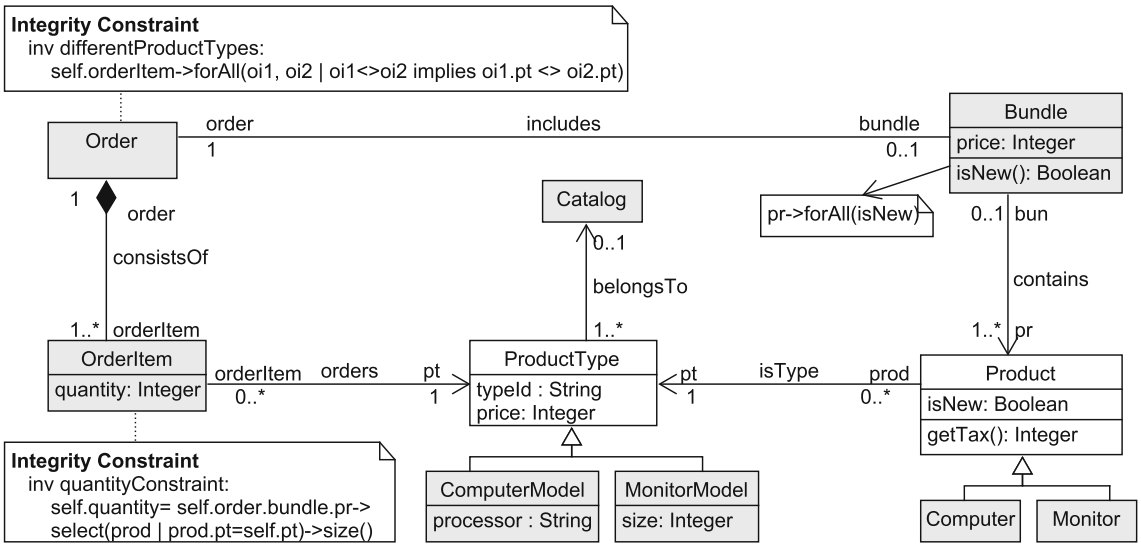



**Fig. 2** The *type-instance* structure from Fig. 1b extended with context that creates an association cycle. Context classes are marked with light gray background

require that the model at level $i$ is a partial instance of the model at level $i + 1$ [11,12].

The engineering argument in favor of MLM claims that the two-level models introduce unnecessary accidental complexity. The class-hierarchy-based model (Fig. 1a) does not capture the notion of a type as an abstract element, independently of its instances, and does not support flexible mechanisms of inheritance and overriding. In addition, dynamic creation of types requires complex mechanisms of dynamic model transformations.

The *type-instance*-based model (Fig. 1b) is conceptually richer, since it explicitly represents the notion of type. But, instances of this model include both objects of type elements and objects of their instances, i.e., an inherent duplication. The subclasses *PCStan, PCDel, MFlat, MCRT* are turned into type instance-objects *pcStan, pcDel, mFlat, mCRT*, whose type identity is described by the *typeId* attribute. This model introduces unnecessary redundancy of type-instance relationships, since instantiation relations are duplicated: There is a built-in instantiation between the elements in a class model and its instance model, and there is an instantiation relation which is expressed by the *isType* association and its links. The instantiation of *pcStan1, pcDel1, mFlat1, mCRT1* is redundant, since it is already captured by their *isType* links. This structure creates models with an excessive amount of redundant elements, as shown in [4,5]. In addition, it mixes types and instances as indistinguishable entities on a single modeling level.

Current studies of multilevel modeling concentrate on the architecture of the types and instances under consideration, while mostly ignoring their context, i.e., the classes, associations, and class-hierarchy structures that surround a *type-instance* structure[1]. However, it appears that contexts do not necessarily fit an ideal multilevel architecture of types and instances. Figure 2 presents a simplified Supply-chain model, following [13–15], that extends the *type-instance* structure from Fig. 1b with a context of an association cycle[2]. The context association cycle of the *type-instance* structure consists of client classes *OrderItem*, *Bundle*, server class *Catalog*, indirect client *Order*, and the dependent class hierarchies of *ProductType* and *Product*. We see that the associated classes cannot be placed in the different levels of *ProductType* and *Product* without crossing the multilevel structure. Moreover, it is difficult to recommend an ideal context-aware multilevel rearchitecture transformation since context structures can vary.

In this paper, we study *context-aware factors* that affect the quality of multilevel models, like *redundancy, level-compositionality, coupling, cohesion*, and *understandability*. The factors are inter-related, sometimes overlapping and sometimes conflicting[3]. We suggest *quantitative syntactic measures* for these factors, like numbers of duplications and of inter-level dependencies. The measures can be used as *metrics* for evaluating accidental complexity in multilevel modeling. The factors and their metrics are used for constructing a set of guidelines for MLM rearchitecting. The idea is that modelers can prioritize the factors according

[1] An exception is a pattern in [5] that suggests a transformation for client classes of a *type-instance* pattern.

[2] Part of this context appears in [4].

[3] Indeed, balancing among conflicting factors while trying to maximize model quality is the essence of software-modeling dialectics. Like the dialectics between the conflicting principles of *high cohesion* and *low coupling*.

to their subjective modeling ideals and use the metrics to build an *MLM quantitative scale* of accidental complexity. We demonstrate our method by constructing our subjective, quantitative, MLM-quality scale, and apply and compare it with the pattern of [5]. This scale is currently under implementation within the multilevel component of our FOML modeling application [16].

The relevance of the context-aware MLM complexity factors and metrics is evaluated in an experimental study of *type-instance* occurrences in multiple real-world models. It was found that most real systems include *type-instance* structures within complex non-trivial contexts that require thoughtful MLM rearchitecture. The contribution of this paper is twofold: (1) study of factors and metrics for context-aware accidental complexity in multilevel models; (2) proposed guidelines and advice for context-aware multilevel rearchitecture of two-level models.

Section 2 describes the MLM paradigm. Section 3 introduces the context-aware factors of MLM accidental complexity with the proposed quantitative measures, and Sect. 4 demonstrates multilevel rearchitecture based on the suggested factors and metrics. Section 5 extends the suggested MLM transformations to apply also to complex contexts with class-hierarchy structures. Section 6 presents the results of the experimental study of real models, and Sect. 7 concludes the paper.

## 2 Background on MLM

MLM emerged from ontological classification of natural domains, like biological ontologies, where classifiers often concentrate on the hierarchical structure of types as entities (i.e., consider the type itself as an entity, independently of its set semantics). A different motivation comes from observing shortcomings of using the popular *Type-Object Pattern* (*TOP*) in software modeling and in object-oriented programming [17–19].

Two early MLM approaches are the *powertype*-based approach [20–22], and the *stereotype*-based approach. Both can be expressed as enriched two-level UML models [23]. In the powertype-based approach, a supertype $S$ of a class-hierarchy structure is associated with a class that denotes its powertype (or a subset of it), i.e., a meta-type whose member objects are the subclasses of $S$. The stereotype-based approach relies on the UML stereotype mechanism to mark classes as domain-specific meta-types.

An axiomatic *First Order Logic* (*FOL*) theory for MLM, entitled *MLT** is presented in [24,25]. This theory defines an ontology-like stratified structure of individuals, types, and meta-types up to three levels. The theory consists of axioms and constraints that define and restrict instantiation levels in a conceptual model, based on the concepts of powertype and categorization, and can support modeling elements like attributes and relationships. Recently, MLT* is compared with the *DeepTelos* theory [26] via a common implementation in the *ConceptBase* deductive system [27,28].

A different approach to MLM, that does not break the two-level framework of OMG, is suggested in [29]. In this approach, termed a *superstructure* for MLM, the essential concepts of MLM that include *typing, instantiation* and *specialization* are specified as a UML/OCL class model. The model, which is implemented in the USE system [30], uses the standard UML elements of *classes, methods, associations, specialization*, and *multiplicity*, with the *redefinition* and *union* inter-association constraints, and with *OCL constraints* and *operations*, to specify the syntax of the MLM domain. This approach shares with MLT* the practice of embedding the MLM specification in a different, already defined, language. Interestingly, both approaches use reduction to SAT solvers, to check soundness [31,32]. Hinkel, in [33], describes a meta-modeling framework for deep modeling, in which the effect of MLM is obtained using structural decomposition and refinements of attributes and references.

The *potency-based approach* of [8,9], which is visualized in the three-level model of Fig. 1c, is based on an explicit level-based syntax. While in the above approaches the level of an element is derived from an MLM specification, in the potency-based approach, a model is specified by explicit declaration of its levels. The levels are merged into a single model by introducing a new linguistic unit termed *clabject* that combines the double role of types as instances of types in a higher level, and as types for objects in lower levels. For example, in Fig. 1c, `PCStan`, `PCDel`, `MFlat`, and `MCRT` in Level 1 are clabjects. Clabjects function as the restricted version of *powertypes* in [21] (captured by the categorization notion of MLT). This conceptualization has much in common with the meta-modeling approach that characterizes *Domain-Specific Modeling Languages* (*DSML*) [6], where the syntax of a DSML singles out the characteristic elements of a domain.

The potency-based approach gains its name from its inherent potency mechanism, for controlling instantiation of model elements along levels. This mechanism has several versions, which enable convenient flexibility of instantiation relations. More recent discussion of different aspects of *potency* in MLM appears in [34,35] and includes finer distinctions of MLM *vitality* notions.

Overall, the potency-based approach breaks the two-level structure of an OMG model, claiming that a sophisticated architecture of levels that recognizes the dual role of types is preferable over a two-level framework, in which types and objects are embedded in a single modeling level. Moreover, MLM prevents redundant duplication, enables dynamic creation of types, and supports flexible attribute inheritance and management.

The semantics of potency-based multilevel modeling is formulated by a *graph-based* theory [10]. In this semantics, multilevel models are viewed as graphs whose nodes represent types and objects, and whose edges stand for the *typed-by* and the *conforms-to* relations. The graph combines the elements of all levels. The levels are implicitly defined by the *typed-by* and the *conforms-to* edges in the graph, and by potency values of nodes. It is unclear how a variety of class model constraints, such as *attributes, qualifiers, and subsetting*, is handled.

Our *mediation-based* set-theoretic MLM framework [11] is built on top of standard class models, using mediators that link class models. A multilevel model consists of multiple modeling *dimensions*, where each dimension is compositionally constructed from successive *level*s. A level consists of a *class model* and an inter-level *mediator*. The mediator of a level is an object model which is a (partial) legal instance of the next higher level (inspired from Herbrand semantics). The class model and the mediator of a level form its *class facet* and *object facet*. Common elements of the class and object facets are the *clabjects* and the *assoclinks* of a level. A level can include *external, inter-level dependencies*, in the form of *potency assignments, inter-links, inter-associations, inter-methods* and *inter-constraints*.

The semantics of an MLM dimension is defined compositionally from legal instances of the class models of the levels, but the instances are restricted to include their level mediators, and to further satisfy the inter-level dependencies of their level. Mediation-based MLM is the only MLM approach in which the *level* notion has a *first-class citizen* status. The class model of a level is an independent component that can be independently developed, verified, tested, validated, and evaluated. Due to its modular component-wise structure, a dimension enjoys flexible management, with simple reuse capabilities.

Mediation-based MLM can be implemented on top of any class modeling tool that can simultaneously support multiple class and object models. This approach is now being implemented as an MLM component in our FOML deductive application [16,36,37], which has this capability (to simultaneously run multiple class and object models). The MLM component uses FOML class modeling as a black box and extends it with inter-level mediators and dependencies.

MLM modeling is implemented in several tools. Potency-based MLM is supported by tools like DeepJava [3], METADEPTH [38], XLM [39], Melanie [40], FMMLx [41,42]. Telos [43] and F-Logic [44] are knowledge representation languages that support unrestricted instance-of and subtyping chains and therefore are natural candidates for MLM. ConceptBase implements Telos [43] in a deductive logic framework, and DeepTelos [45] and [26] extend it to support multilevel reasoning. Industrial applications that rely on the MLM capabilities of F-logic are described in [46–48].

De Lara et al., in [5], present several patterns for rearchitecting two-level modeling structures into multilevel modeling. The patterns focus on improving the modeling of type-instance structures and of class hierarchies by transformation into multilevel designs. The patterns refer to some context aspects, but the impact on context classes is not investigated. Table 1 summarizes the main MLM terminology. For each notion, we provide a textual explanation and a visual description when relevant.

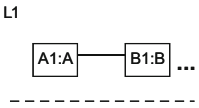
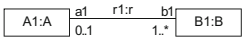
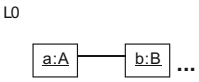
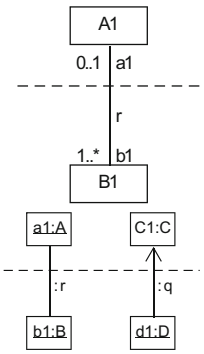
The notion of *Accidental complexity* was introduced in [2] in order to distinguish between *essential complexity*, which characterizes a problem domain, to accidental one, that results from using weak implementation languages, weak abstractions, or bad design. This notion is frequently used in companion with a variety of software metrics, and also with respect to efficiency of concrete application [49]. In MLM, the claim is that the explicit modeling of instantiation reduces accidental complexity since it reduces redundancy and improves conceptualization. The thesis of [50], which constructs a broad multilevel framework, compares the presented framework with two level modeling, using accidental complexity and several standard OO metrics. Recently, [51] suggests metrics based on instantiation and specialization structure, for comparing multilevel representations (including two level ones).

Our initial study of context of *type-instance* structures [52] singled out few context-dependent structures that affect the quality of a multilevel model and suggested some measures. This paper presents continuation, elaboration, and broadening of the initial structure: It suggests *context-aware factors* that affect the conceptual quality and management of a multilevel model, introduces *metrics* for measuring these factors, and provides *guidelines* for using these factors in MLM representation decisions. The relevance of the context-aware analysis of representational factors is evaluated by an experimental study of their actual occurrence in complex contexts of conceptual models.

The factors and metrics are defined in terms of syntactic features of multilevel models, i.e., classes, objects, attributes, levels, inter-relationships like associations, links, instantiation, specialization, potency values (if exist), and their characterization as intra- or inter-level. The definitions apply to all MLM approaches, independently of whether they support an explicitly declared or a computed *level* feature.

We adopt the conventional visual description for multilevel models, where levels are marked as horizontal layers (as in Fig. 1c), classes, objects, associations, links, and specializations are visualized as in class and object diagrams, and clabjects and assoclinks have combined visualizations that capture their double roles. The visualization of these syntactic MLM elements is shown in Fig. 3.

**Table 1** Basic MLM terminology

| Term | Description | Visualization |
|---|---|---|
| *Level* | A level stands for a collection of modeling elements from Class and Object models. Levels are marked by numeric values. Elements of a level can be instances of elements in higher levels. |   |
| *Clabject* | A class which is an instance of a class on a higher-level. |   |
| Clabject *class-facet* | Clabject as a class | |
| Clabject *object-facet* | Clabject as a class-instance object | |
| *Assoclink* | An association which is a link of an association on a higher-level. |   |
| *Potency* | A numeric assignment to level elements. It specifies the number of successive instantiations (or inheritance for attributes) of an element. Exact definition varies. | @ 2 |
| *Leap potency* | A numeric assignment to level elements. It specifies instantiation with a leap, $n$ levels below. Exact definition varies. | @ (2) |
| *Object level* | The lowest level in a multilevel sequence of levels. Its elements are objects and links. |   |
| *Inter-level dependency* | A dependency between elements from different, possibly unadjacent, levels: associations, links, methods constraints. | |
| *Inter-level association* | An association between classes from different levels |   |
| *Inter-level link* | A link between objects or clabjects from different levels |   |
| *Inter-level method* | A method code that references an element from another level | |
| *Inter-level constraint* | A constraint that references an element from another level | |

Visualization of entities distinguishes between *objects, classes*, and *clabjects*. Objects do not have a class role, classes do not have an object role, and clabjects have a double role: Classes and objects. In the implicit level specification approaches, the objects are leaves of specialization–instantiation chains and classes are roots of such chains. Clabjects are intermediate nodes. In the potency-based approaches, a 0-valued member of a class is an object, but not a class, since the 0-valued potency stops the instantiation chain.[4]

Visualization of relations between entities distinguishes between *links, associations*, and *assoclinks*. Links are instances of associations and do not have an association role, associations do not have a link role, and assoclinks have a double role: Associations and links. A link can relate two objects, or an object with a clabject, or even two clabjects. A link with a clabject refers only to the object facet of the clabject. In the potency-based approaches, a 0-valued instance of an association must be a link. A relationship between enti-

[4] Some publications assume that 0-potency value of a "class" $C$ points that $C$ is an abstract class. This situation contradicts the declared potency intention, since the set semantics of classes implies that all instances of

Footnote 4 continued
subclasses of $C$ are also instances of $C$. 0-potency for a class member means that it belongs only to the object facet of a level, and not to its class facet.

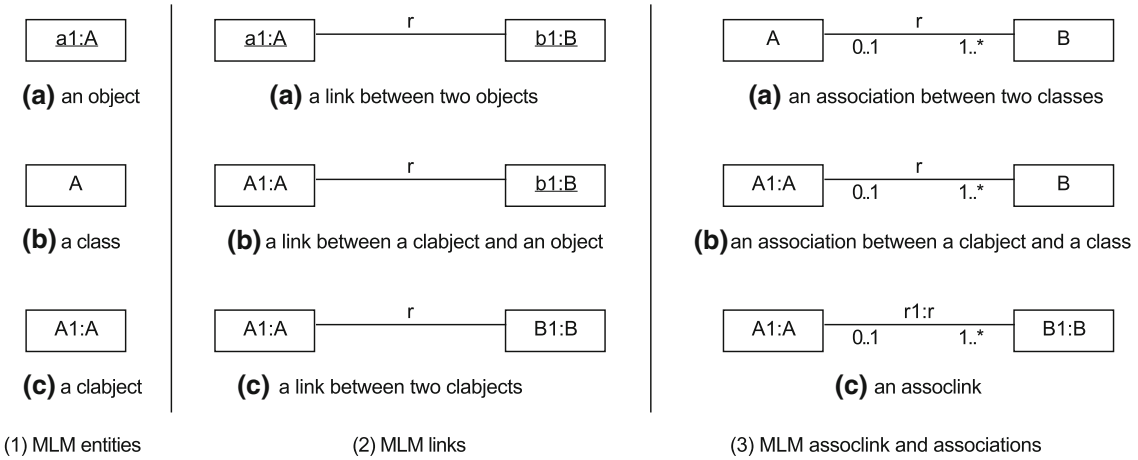



**Fig. 3** Visualization of MLM entities and relations

ties that reside on two different levels is *inter-level*, i.e., an *inter-association* or *inter-assoclink* or *inter-link*.

## 3 Factors and metrics for evaluating accidental complexity in MLM

The term *Accidental Complexity* (*AC*) was first used by Brooks [2] as a way to capture non-essential complexity, caused by a language or a tool for implementation, and is not an essential characteristic of the solved problem. This concept has been used in the study of MLM for arguing that MLM reduces accidental complexity of modeling *type-instance* relationships [4,5]. The argument is, roughly, that MLM directly models the problem domain and avoids needless duplication. However, when surrounding context in a model is considered, MLM rearchitecting might not be straightforward, and there are multiple alternatives, each with its own pros and cons.

In this section, we identify *context-aware factors* of multilevel models, which might affect their complexity in different respects. The factors are defined in terms of general MLM terminology, like *levels, instantiation, subtyping, association*, and avoid approach-specific terminology like *potency* marking, *categorization*, or *level-mediation*. Hence, they are general to all MLM approaches. Note that the *level* notion exists in all approaches: In some representations, it is explicit in the syntax, while in others it is implicitly computed.

For most factors, we suggest quantitative measures for numerical estimation of the "amount" of factor occurrence. The quantitative measures form a set of metrics for evaluating the quality of multilevel models. The factors and measures are motivated and demonstrated on the simplified Supply-chain model from Fig. 2 that includes a context of an association cycle. Examples of factors and metrics that deal with inter-level dependencies might include data objects on Level L0 that are interlinked with elements on higher levels. Contexts with class-hierarchy structures are introduced later on in the paper, in Sect. 5.

### 3.1 Redundancy

Redundancy can be measured by *duplication* of elements. This is the main argument for the accidental complexity in the *type-instance* structure in Fig. 1b. The claim is that the instantiation relation is duplicated, and the instance classes *Computer* and *Monitor* are redundant. The multilevel model in Fig. 1c is considerably smaller since instances of a type-class are combined with their instance-classes, into *clabjects*.

Redundancy can still arise in a multilevel rearchitecture of the context, usually, when a superclass is removed, in favor of direct instantiation of a type class in a higher level, as in Fig. 1c. In that case, the remodeling of the context can create redundancy and stability problems.

#### Redundancy due to a missing abstraction barrier:

Consider the MLM remodeling in Fig. 1c. The *Product* class-hierarchy is removed from Level 1, in favor of four concrete instance clabjects of *ProductType*, so to remove repeated characterization via instantiation and class specialization. But, this MLM architecture removes an abstraction barrier, implying loss of visibility of the subclasses via the removed superclass. Therefore, previously inherited client-access features must be provided for each subclass. Figures 4 and 5 show two cases of such redundancy.

In Fig. 4, Level L1 includes four instance clabjects of *ProductType*: *PCDel, PCStan, MFlat, and MCRT*, and the *Product* class hierarchy is removed. Client class *Bundle* of *Product* in Fig. 2 is also positioned on Level L1. Since the *abstraction barrier* superclass *Product* is removed, association *contains* is duplicated four times, and method *isNew*

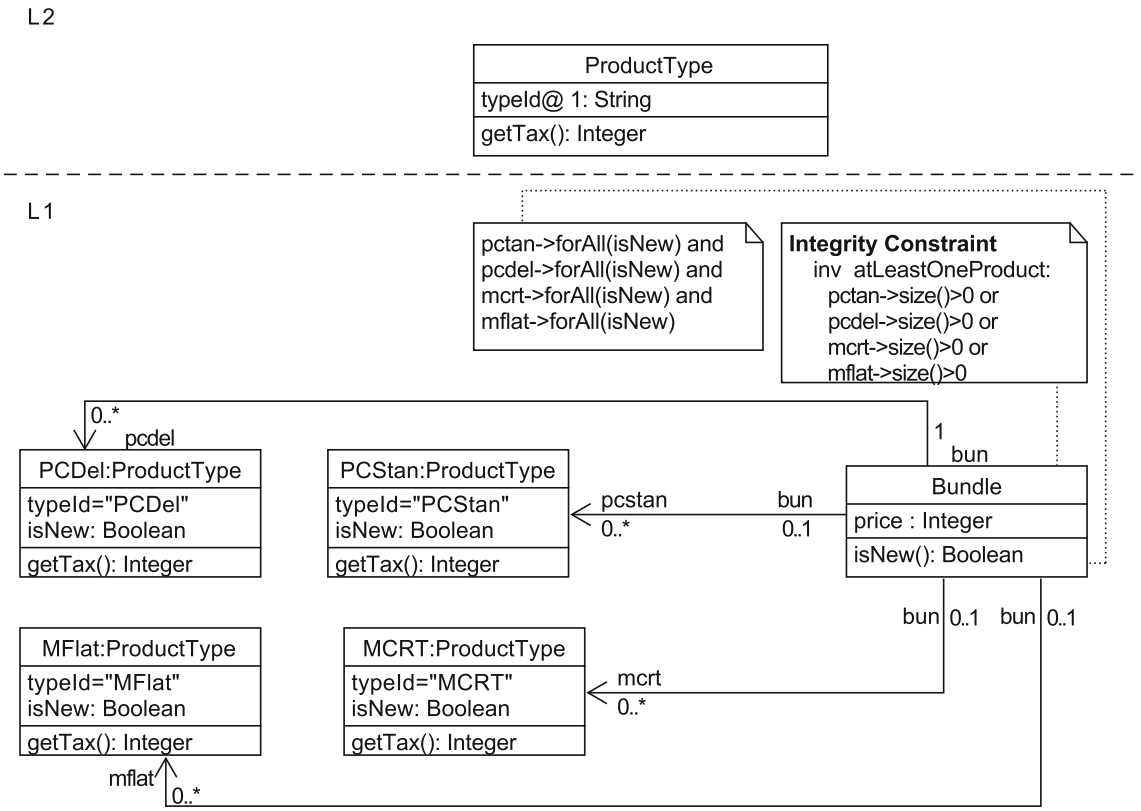



**Fig. 4** Duplication of client associations and method code due to a lost superclass

of *Bundle* needs to iterate over the four associations, with duplicated code fragments. Moreover, a new constraint, *atLeastOneProd*, is needed, to ensure that a bundle is not empty. The overall duplication toll is 6, as there are 3 redundant associations and 3 duplicated code fragments.

Figure 5 shows a different case, where a missing abstraction barrier causes redundancy in instantiation of a class on a higher level. In this case, client class *OrderItem* of *ProductType* in Fig. 2 is positioned on Level L2 (as suggested in [5], see Fig. 10). In addition, there is an integrity constraint that puts an upper bound on values of the *quantity* attribute of instance clabjects of *OrderItem* that reference an instance clabject of *ComputerModel*. On Level L1, since the *Computer* super class is removed, the instance clabjects of *OrderItem* must be duplicated for their referenced *Product* clabjects (since the *pt* property on Level L2 has multiplicity 1), and the constraint must be duplicated for the two *OrderItem* clabjects, *OiPCDel, OiPCStan*, that reference clabjects *PCDel, PCStan*. The duplication toll is 4, as there is one clabject duplication, 2 association duplications, and one constraint duplication. In general, the removal of an abstraction barrier causes also stability problems, which are discussed in Subsect. 3.2.

### Redundancy of singletons

The traditional handling of singletons in object-oriented software involves inherent redundancy: A single member object of the singleton class represents the class. The representative object is redundant, as its content duplicates the class content. In MLM, where classes and objects can be unified into clabjects, singletons create unnecessary redundancy. Singletons can be viewed as a special kind of clabjects, having themselves as their only instances. References to singletons are links to clabjects, i.e., *object-clabject* or *clabject-clabject* links, as shown in Fig. 3. Sure enough, this might create cross-level dependencies, and it is up to a modeler to determine priorities: duplication or potential cross-level relationships.

Figure 6 shows these options. Figure 6a, shows the duplication caused by creating the representative *Catalog* object, while Fig. 6b shows the cross-level dependency, created by the *getTax* methods of the *ProductType* instance clabjects, that trigger the *computeTax* method of the *Catalog* class on Level L2.

### Refinement

Sometimes, depending on subject domain and concrete requirements, an unnecessary repetition turns into a desirable

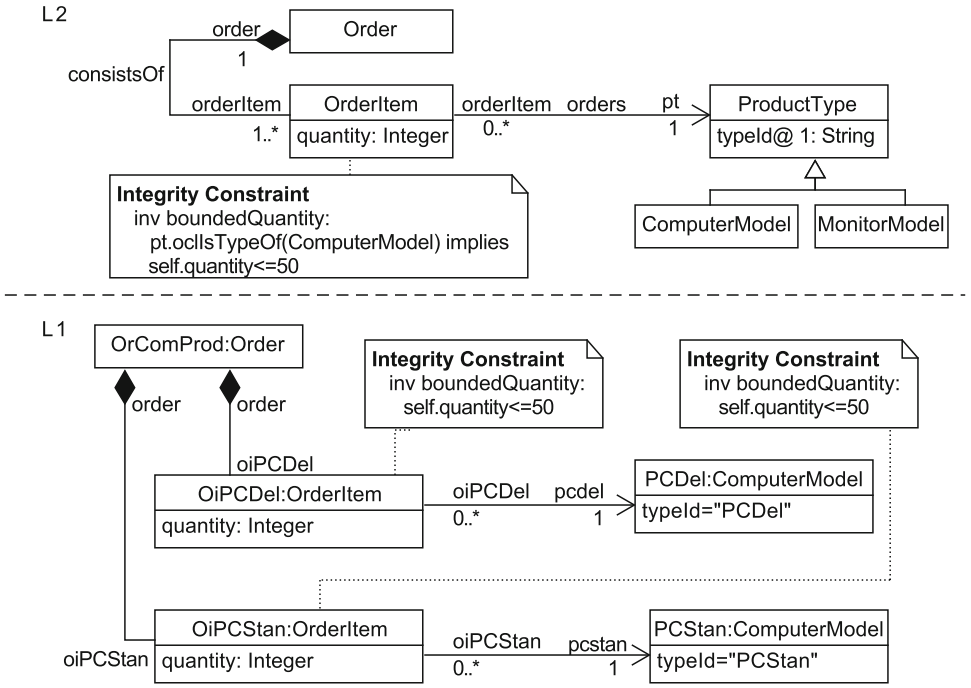



**Fig. 5** Duplication of clabjects and assoclinks on Level L1 due to lost superclass and a multiplicity constraint on Level L2

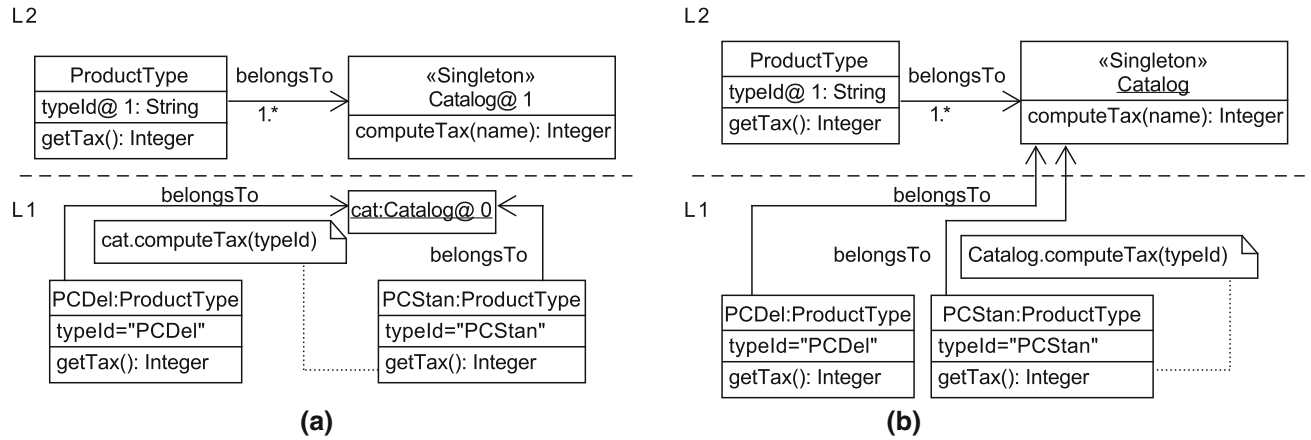



**Fig. 6** Singleton duplication vs. inter-level dependencies

refinement. For example, in Fig. 7, client class *Bundle* from Fig. 2 is positioned on Level L2 and has two *Bundle* instance clabjects on Level L1, *BundleDel* and *BundleStan*, each with its own bundle regulations. We see that the redundancy of the *OrderItem* client clabjects in Fig. 5 turns into a desirable refinement that enables distinction between two kinds of bundles. Overall, isomorphic MLM architectures might yield undesirable redundancy in one model, while enabling desirable refinement in another model.

## 3.2 Level instability

A software model is stable if a local change does not cause propagation of multiple local changes, or some global changes. Model stability is an evolution aspect that reflects *dependency (coupling) relationships* among model elements. In general, lower coupling implies greater stability. In MLM, there is an *inherent (built-in) level coupling* between adjacent levels, since the instance facet of a level class model is a partial instance of its immediate higher level (see [11], and also Sect. 2). Hence, lower levels always depend on higher levels: Changes in a level class model affect its direct and indirect instances.

Our discussion of MLM instability identifies typical syntactic coupling-structures that might increase the inherent MLM level coupling, such as (1) inter-level dependencies like inter-level associations and links, inter-level method calls

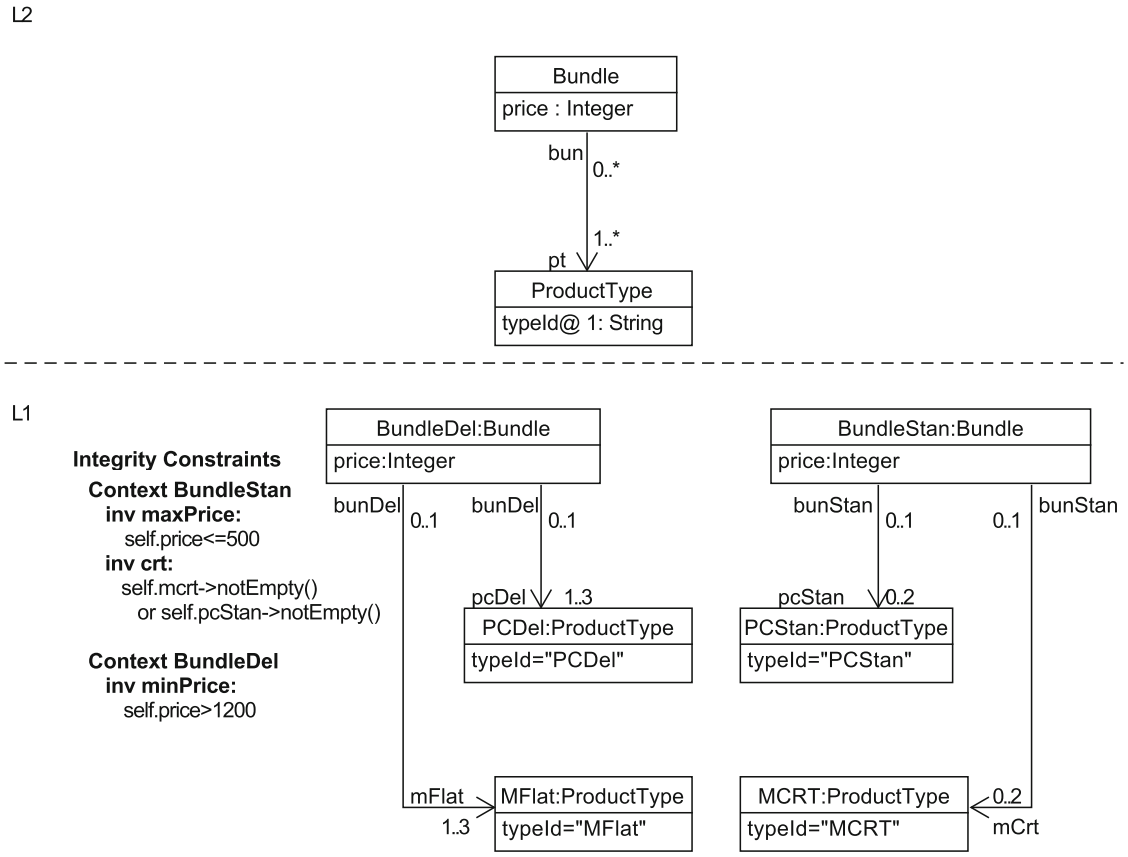



**Fig. 7** Refinement of clabjects and assoclinks

(as in Fig. 6b) and inter-level constraints; (2) explicit type-object dependencies; and (3) missing abstraction barriers.

### Upward and downward level-coupling

*Upward level-coupling* characterizes cases where a level might be affected by changes in multiple higher levels.[5] This factor might be caused by a variety of upward inter-level dependencies. For example, assignment of leap potency to a class that is associated (directly or not) with a class with a smaller potency value creates upward level coupling. To see this, consider again Fig. 2 and the pattern of de Lara et al. [5] in Fig. 10 that suggests putting client classes of *Product* on Level L2, with a *leap potency* value 2.

Following this pattern, *Bundle* is on Level L2 of the rearchitected MLM model, with leap potency 2, e.g., *Bundle@(2)*, and *Bundle* and *contains* are instantiated directly on Level L0, and *bundle*s are linked to concrete monitors or computers in Level L0.

If class *Order* is placed on Level L2 with potency 1 (like *OrderItem*), its instances reside on Level L1, and *bundles* on Level 0 are linked to *orders* on Level L1, implying that a state of Level L0 instantiates multiple classes in Levels L1 and L2. A change in either of these levels might affect Level 0. If *Order* on Level L2 has, like *Bundle*, leap potency 2, then the *order* of a *bundle* resides on level 0, but its *order-items* reside on Level 1, implying again, dependency of Level L0 on Levels L1 and L2. These remodeling options are shown in Fig. 8.

Figure 9 shows how changes in Levels L1 and L2 of a model based on Fig. 8a affect Level L0. Evolution is marked by a cross-out line in the left diagram and a correction in the right one. On Level L2, a previous maximum multiplicity value of 2 for the *bundle* property is modified to 1. The change implies removal of at least one *Bundle* object from Level L0. On Level 1, the type of the "id" attribute of the *PCDell* clabject has changed from `Integer` to `String`, implying a necessary change in the attribute value of the *PCDel1* object on Level L0.

*Downward level-coupling* is dual to upward level-coupling. It characterizes cases where a change in a single level might affect multiple lower levels (reminds the dual software smell

[5] Reminds the Divergent Change software smell [53] that describes cases where a software element can be affected by changes in multiple places in the code.

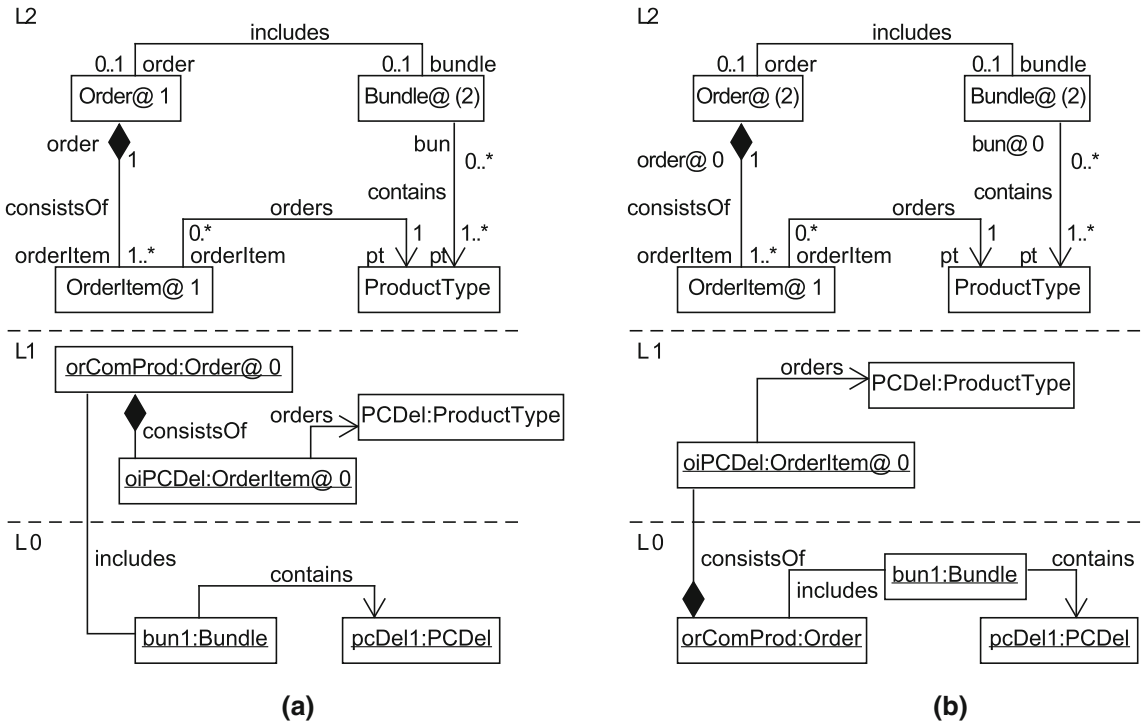



**Fig. 8** MLM models with an upward level coupling

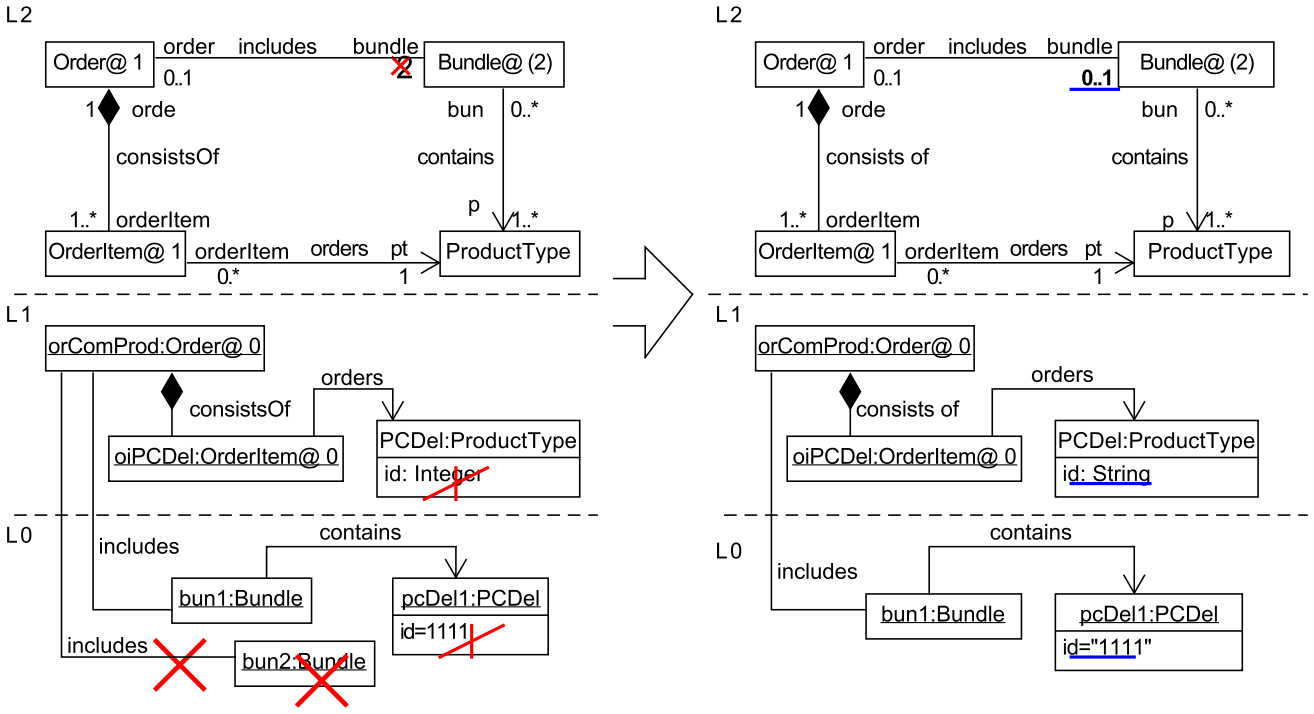



**Fig. 9** Level instability due to upward level coupling

to Divergent Change[6]). For example, in the models in Fig. 8, changes to elements in Level L2 might affect both Levels L1 and L0. Upward and downward level-couplings are not necessarily symmetric, since an inter-level dependency might be single directed, like directed associations or method calls.

[6] Shotgun Surgery is the software smell characterizing cases where a change in one place implies multiple changes in the code.

Altogether, possible metrics for upward or downward level-coupling are (1) existence of association sequences between classes with different (leap) potency values; (2) explicit inter-level dependencies—associations, links, method calls, constraints.

#### Type-object mixture

The dynamics of change of (what is considered as) "data" or "objects" is usually way more intensive than change of types. For example, in the simplistic Supply-chain model of Fig. 2, concrete products, bundles, orders and order-items, constantly change between states, while product types usually stay stable, e.g., while new computer brands are not frequently introduced, new computers and monitors are bought on a daily or even by the hour basis. Indeed, in MLM the unification of classes and objects as clabjects invites also type dynamics, but still the expected pace of object-change is much higher.

Type-object mixture can create level instability due to intensive object changes. For example, in Fig. 8a and 8b, *oiPCDel* on Level 1, which is an instance of *OrderItem* on Level L2, is linked, directly or indirectly, to data orders and bundles on Level L0. Hence, changes on Level L0 imply changes on Level 1, implying constant instability of Level L1, which is a level of types.

Type-object mixture can be identified by levels with mixture of types and objects, and by the level-instability metrics that were described above, mainly explicit type-object dependency forms and associations between classes with different (leap) potency values.

#### Missing abstraction barrier

Figure 4 shows duplication of associations and code fragments, due to the loss of the abstract superclass *Product*. Hence, the *Bundle* class tightly depends on changes in the concrete *Product* clabjects: Addition or removal of a *Product* clabject implies addition or removal of an association and changes in the code of the *isNew* method and the *atLeastOneProduct* constraint. This tight dependency increases the instability of Level L1.

#### MLM Compositionality

Compositionality is a desirable feature in software, since the syntax, semantics, management, and understanding of a composite code/model can emerge from those of the components. In MLM, where a model can be composed of levels, compositionality depends on the approach and on the amount of inter-level dependencies. The MLM approaches in [25,26] are not syntactically compositional; potency-based approaches as in [8,38,41] assume a level-based compositional syntax; and the mediation-based framework in [11] has also level-based compositional semantics and management. Metrics for compositionality are those of the upward and downward coupling factors, i.e., syntax forms that imply inter-level dependencies (beyond the built-in inherent level-dependency in MLM): Reduced inter-level dependency implies increased compositionality.

### 3.3 Understandability, conceptualization, and client visibility

*Understandability* and *conceptualization* are qualitative factors of model quality that to a great extent depend on subjective judgment of users. *Understandability* refers to the capability of users to estimate the suitability of models to requirements, and *conceptualization* refers to the ability of a model to capture and account for relevant domain concepts. Both factors are highly important for maintenance and reuse but difficult to estimate, since they depend on cognitive capabilities and experience of users. *Client visibility* is a comparative factor that reflects existence of abstraction barriers like class hierarchies and interfaces.

#### Understandability

Understandability and readability of software models have been intensively studied [54,55]. These factors are affected by size and structure of models, as well as by usage of modeling elements with complex or unclear semantics [55–59]. MLM introduces *instantiation* as a new modeling element, and its syntactic and semantic integration with other standard OO modeling elements poses a new understandability challenge. It was already noted that the view of a level in a multilevel architecture should be restricted to three, i.e., data, model, meta-model [47]. Our cooperation with the developers of the Ink language [60] shows that working with MLM requires appropriate training.

The clarity and understandability of a multilevel model are affected by syntactical features like the *number of levels* and *number of constraints*. Explicit intra-level or inter-level visual dependencies, e.g., multiplicity constraints on associations, are clearer than implicit text constraints or implied dependencies, e.g., level dependency via leap potency.

*Cohesion* is a software quality factor that estimates how much elements in a component (class, module) belong or relate to each other. It reflects the strength of the component as a conceptual unit. *High cohesion* improves robustness, reliability, reusability and understandability. In MLM, where the *type-instance* relation is built into the level architecture, it seems that *mixture of types and objects in a single level* increases *incohesion*. For example, in the two versions of Fig. 8, Level L2 includes *ProductType*, with its types-of-types semantics, together with classes *Bundle*, *Order*, and

*OrderItem* which have simple type semantics. This creates semantic incohesion which reflects on lower levels. In Level L1 of both versions, the object *oiPCDel* (not a clabject) is linked to the *PCDel* class. This mixture creates *level incohesion* that causes other problems of coupling, compositionality and instability.

Suggested metrics for understandability include *number of levels, number of non-built-in constraints, number of inter-level dependencies*, and *mixture of potency values in a level*, which is likely to introduce level incohesion.

#### Conceptualization and client visibility

MLM supports the *type-instance* relationship as a built-in, *first class citizen* concept, and enables flexible control of *attribute* and *association inheritance* over instantiation and specialization hierarchies. In standard OO modeling, attribute inheritance can be stopped only by using static attributes, and association inheritance can be overridden via mechanisms like the UML *redefinition constraint*, which does not remove the unused overridden association. In contrast, MLM attribute and association inheritance can be stopped by the potency mechanism.

On the other hand, MLM rearchitecture often removes superclasses, in order to avoid duplication of inheritance via instantiation and specialization. But this step might remove necessary abstractions, and reduce client visibility. We have already seen that it can cause increased redundancy and level instability. Altogether, MLM might increase conceptualization in one direction, while reducing abstraction and client visibility in another.

### 3.4 Metrics for quantitative evaluation of the identified MLM complexity factors

In this subsection, we define possible metrics for measuring the MLM complexity factors. In order to enable comparison and combination of metrics, we suggest possible normalization computations, that eliminate the impact of overall size parameters of models, and shift the value ranges of metrics to the [0..1] interval. Normalization rules are metric specific.

Some metrics refer to potency marking, due to its popularity, and are relevant only in potency-based approaches. In addition, although no metric refers to the L0 data level in MLM, inter-level dependencies between data objects in an L0 instance and elements in higher levels in the model might affect the overall accidental complexity evaluation.

1. **Duplication metrics:** Intended to measure redundancy of modeling elements in a level. Possible normalization is by dividing the number of counted elements by the maximal number of such elements in a level.
   (a) *Number of duplicated classes or objects in a level* ($\#dup\_clsObj$): In Fig. 5, $\#dup\_clsObj(L1) = \frac{1}{5}$, and over the two level model, it is $\#dup\_clsObj(L1\_L2) = \frac{0+1}{5+5} = 0.1$.
   (b) *Number of duplicated associations or links in a level* ($\#dup\_assocLnk$): In Fig. 4 $\#dup\_assocLnk(L1) = \frac{3}{4}$, and in Fig. 5 $\#dup\_assocLnk(L1) = \frac{2}{4} = 0.5$.
   (c) *Number of duplicated attributes* ($\#dup\_att$): Attribute duplication is a byproduct of class duplication, but it deserves a separate metric because duplication of classes with multiple attributes has a higher redundancy rate. In Fig. 5 $\#dup\_att(L1) = \frac{1}{4}$.
   (d) *Number of duplicated methods or constraints* ($\#dup\_methConst$): In Fig. 4 $\#dup\_methConst(L1) = \frac{3}{6} = 0.5$, and in Fig. 5 $\#dup\_methConst(L1) = \frac{1}{2}$.
   (e) *Number of duplicated method or constraint fragments* ($\#dup\_methConstFrag$): In Fig. 4 $\#dup\_methConstFrag(L1) = \frac{3+3}{4+4} = 0.75$.
   (f) *Number of singleton class-object duplications* ($\#dup\_sglTClsObj$): Measures redundancy due to duplication of singleton classes by their single objects. Possible normalization is by dividing by the overall number of classes in a model. In Fig. 6a, $\#dup\_sglTClsObj(L1\_L2) = \frac{1}{4}$.
2. **Level instability metrics:** The metrics in this category identify relationships between elements that differ in some aspects, and the difference causes model instability. Normalization rules are metric specific.
   (a) *Number of intra-level associations between classes with different potency values* ($\#diffPot\_assoc$): This metric can just count such associations, or also consider the difference in potency values. Leap potency increases the instability. Possible normalization is by dividing by the overall number of associations in a level. In Fig. 8a, $\#diffPot\_assoc(L2) = \frac{3}{4}$.
   (b) *Number of inter-level dependencies* ($\#intL\_assoc$, $\#intL\_lnk$, $\#intL\_methConst$): Can be normalized using division by the overall number of elements of the same kind in the model. In Fig. 8a, $\#intL\_lnk(L0\_L1\_L2) = \frac{1}{8}$ (normalized over associations and links), and in Fig. 6b, $\#intL\_assoc(L1\_L2) = \frac{2}{3}$, and $\#intL\_methConst(L1\_L2) = \frac{2}{4} = 0.5$.
   (c) *Number of inter-level dependencies of a class* ($\#intL\_clsDep$): This metric measures class instability. It can be normalized over the overall number of class dependencies. In Fig. 6b, $\#intL\_clsDep(L1.PCDel) = \frac{2}{2} = 1$.
   (d) *Object-type mixture in a level* ($mx\_objTp$) or *Number of object-clabject links in a level* ($\#lnk\_objTp$): $mx\_objTp$ can be normalized over the number of levels in a model, and $\#lnk\_objTp$ can be normalized over the overall number of associations and links in

a level. In Fig. 8a, $mx\_objTp(L0\_L1\_L2) = \frac{1}{3}$ and $\#lnk\_objTp(L1) = \frac{1}{2}$.

(e) *Maximal potency difference in a level* ($pt\_diff$): This is also a metric for understandability: Greater potency difference in a level decreases its understandability. $pt\_diff$ can be normalized over the maximal potency of a level. In Fig. 8a, $pt\_diff(L2) = \frac{1}{2}$.

3. **Understandability metrics**:

(a) *Number-of-levels-in-a-model* ($\#lvls$): The clarity of the intended semantics of an element rapidly decreases as the number of levels below its level increases.

(b) *Number of classes with leap potency in a level* ($\#cls\_lpPot$): Leap potency decreases understandability, since it causes irregular dependency between non-adjacent levels. $\#cls\_lpPot$ can be normalized over the number of classes in a level. In Fig. 8b, $\#cls\_lpPot(L2) = \frac{2}{4} = 0.5$.

In addition to the MLM metrics above, there are, of course, standard OO accidental complexity metrics like dependency of a class on multiple other classes, or associated textual (non-visual) constraints.

## 4 MLM rearchitecture based on the accidental complexity factors and metrics

This section concentrates on the role that the accidental complexity factors and metrics can play in the construction of multilevel models. We already noted that factors might have conflicting effects on accidental complexity. For example, level cohesion and understandability might reduce MLM stability and MLM restructuring might increase duplication of context classes and associations. Our approach is that a modeler should first determine his/her *modeling ideals* and develop a multilevel model in light of these decisions, with quantitative estimation of the resulting accidental complexity.

We present two alternative MLM transformations for the two-level *type-instance*-based model in Fig. 2 and analyze and quantify the resulting accidental-complexity metrics. The context-aware analysis of accidental complexity in MLM is mainly comparative. The MLM body of real systems is still evolving and is not sufficiently rich to present clear evidence and support for absolute estimation of high or low complexity. Therefore, the quantitative evaluation of accidental complexity should be considered comparatively. This section helps in comparing the accidental-complexity values of various metrics, following the two MLM transformations.

### 4.1 MLM rearchitecture following De Lara et al. [5]

De Lara et al. present a pattern for multilevel rearchitecting of client classes of a *type-instance* structure, as shown in Fig. 10.

The pattern puts all client classes in level 2, i.e., the level of types of types, but with different *potency* kinds and values. Client classes of the *Type* class receive a standard potency of 1, while client classes of the *Instance* class receive a *leap potency* value of 2. Therefore, *TypeLevelClass* is instantiated on Level 1, while *InstanceLevelClass* is instantiated on Level 0.

Figure 11 shows an MLM rearchitecture of Fig. 2, following this pattern.[7]

Class *OrderItem*, being a client class of *ProductType*, appears on Level 2, the level of types of types, with potency value 1. Class *Bundle* appears on Level 2 with leap potency (2), and class *Order* is (arbitrarily) positioned on Level 2, with *OrderItem*. Level 1, the level of types, includes 4 instance clabjects of *ProductType*, with two instances of *OrderItem* (the *orders* association requires at most a single instance of *OrderItem* for each) and one instance of *Order*. *Bundle* is instantiated on Level 0 (leap potency 2), and according to [5], the potency 0 of the *pr* property of the *contains* association marks instantiation on Level 0.[8] Therefore, the *Bundle* instance on Level 0 is linked to the instances of the *Product* clabjects, and to the *Order* object on Level 1. Note that the latter is a simple kind of *inter-link* that relates objects on adjacent levels.

Analysis of accidental complexity in the MLM rearchitecture of Fig. 11 points to occurrence of three accidental-complexity factors which are summarized below, and quantified by the metrics in Table 2.

1. **Redundancy:** Method `getTax()` is duplicated four times in the concrete *Product* clabjects on Level 1.
2. **Level instability:** There is upward and downward coupling due to a variety of potency values on Level 2, and objects on Level 0 that instantiate classes on multiple levels. There is a compositionality problem due to the leap potency on Level 2 and the inter-level links, and there is an object-type mixture on Level 1.
3. **Conceptualization and Understandability:** The leap potency creates an understandability problem. The inappropriate placement of *Order, OrderItem*, and *Bundle* on Level 2 (types of types), and the object-type mixture on Level 1, creates level incohesion.

[7] The integrity constraint `quantityConstraint` on the *OrderItem* class is removed since it requires an MLM constraint language [11,16, 61].

[8] Indeed, an ambiguous semantics of the potency labeling.

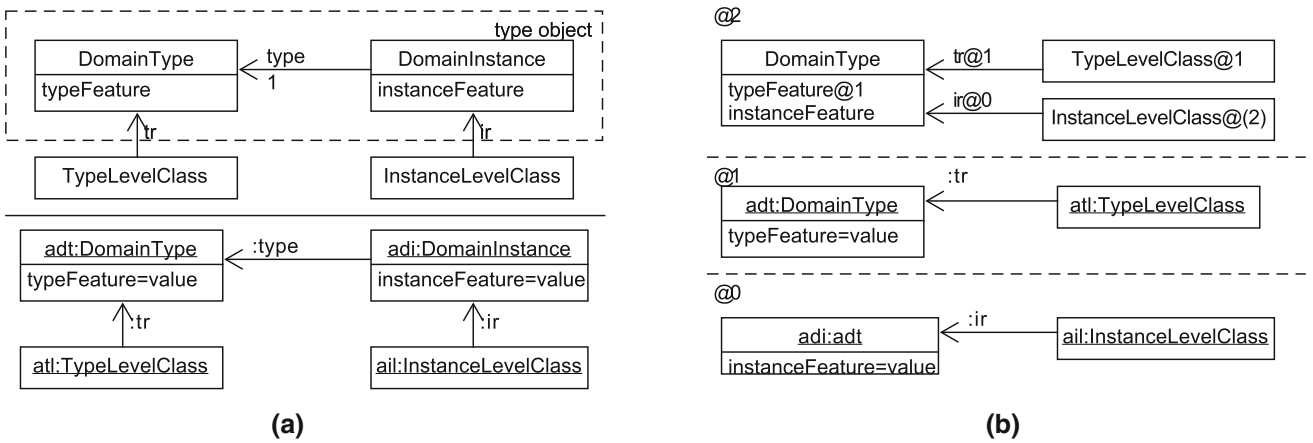



**Fig. 10** The pattern of [5] for MLM rearchitecting of a type-instance structure with context

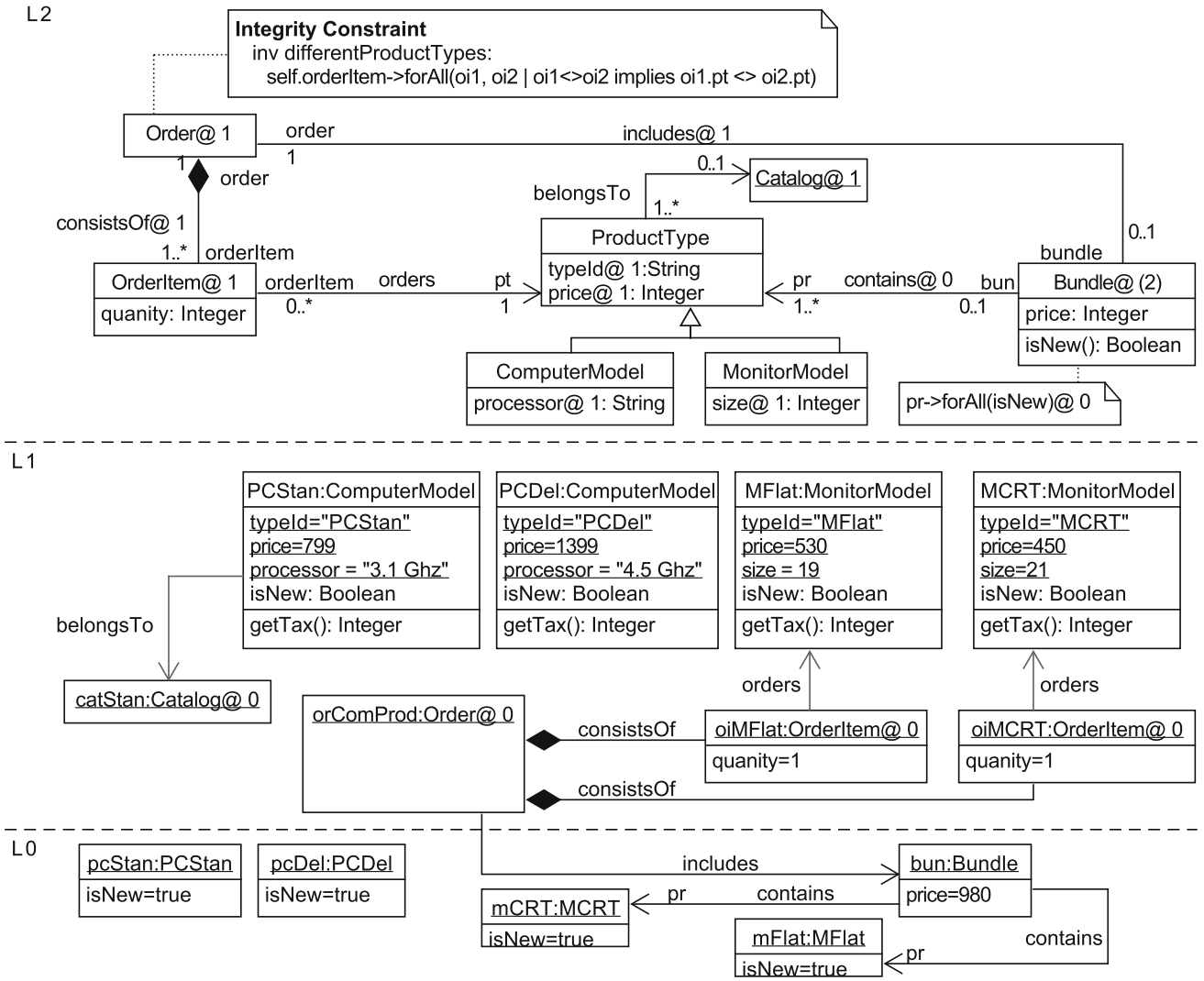



**Fig. 11** Application of the *type-instance* rearchitecting pattern of [5] (appears in Fig. 10) to the *type-instance* structure in Fig. 2

**Table 2** Accidental complexity in Fig. 11

| Metric | Value |
|---|---|
| $\#duplicatedmethodsorconstraints$ | $\#dup\_methConst(L1) = \frac{3}{4}$ |
| $\#intra-levelassociationsbetween$ $classeswithdifferentpotencyvalues$ | $\#diffPot\_assoc(L1\_L2) = \frac{3}{4}$ |
| $\#inter-levellinks$ | $\#intL\_lnk(L0\_L1\_L2) = \frac{1}{8}$ |
| $\#object-clabjectlinksinalevel$ | $\#lnk\_objTp(L1) = \frac{3}{5}$ |
| $Maximalpotencydifferenceinalevel$ | $pt\_diff(L2) = \frac{1}{2}$ |
| $\#classeswithleappotencyinalevel$ | $\#cls\_lpPot(L2) = \frac{1}{7}$ |

## 4.2 Context-aware MLM rearchitecture

Context classes of a *type-instance* structure might be interrelated in complex and challenging ways. In complex contexts, every multilevel rearchitecture of context classes might reveal some accidental-complexity factors, while avoiding others. There is no *silver bullet* transformation that minimizes all accidental-complexity factors. Modelers should determine their ideals and try to reduce accidental complexity of their goal factors.

Our context-aware MLM rearchitecture advice follows two directives:

1. *Conceptualization* and *Understandability*, with emphasis on *level cohesion*: These factors are essential for reliability, robustness, and reuse. In order to follow these factors, the MLM transformation determines the level of classes by their *intended semantics*, rather than by their context associations. This approach is based on a faithful view of reality and yields level cohesion. The level of a class is determined by the question: "What is its semantically intended level," i.e., Level 2, the level of types of types, or Level 1, the level of types of objects, or Level 0, the level of objects.
2. *Client visibility*: This factor is essential for achieving abstraction and encapsulation and affects all accidental-complexity factors. Therefore, breaking a class-hierarchy structure between several levels should be carefully considered, and the impact on client classes should be studied. Figure 12 shows the MLM transformation advice for class-hierarchy structures that have client classes:

Following these ideals, Classes *OrderItem, Order, Bundle*, in Fig. 13, are positioned on Level 1, since they have an intended semantics of types. Class *Product* is not removed (as in Fig. 1c), but appears on Level 1, as a superclass of the four *Product* clabjects. These placements increase understandability and visibility but add an inter-level link.[9]

Class *Catalog* is positioned on level 2 with potency 1, since it stands for a set of product models and does not have two levels of instantiations. Indeed, it is associated only with types of products, and not related to concrete products. Its instance objects on Level 1 are linked to product types, thereby creating mixture of object and types incohesion. But as catalogs are not linked to objects on Level 0, there is no level instability.

Analysis of accidental complexity in the MLM rearchitecture of Fig. 13 points to occurrence of two accidental-complexity factors which are summarized below, and quantified by the metrics in Table 3.

1. *Redundancy:* The inclusion of the *Product* superclass on Level 1 introduces the duplication that MLM modelers try to avoid: The concrete product clabjects specialize *Product* and are also instance clabjects of *ProductType* subclasses.
2. *Level instability:* The only source of instability is the inter-level links from *OrderItem* on Level 1 to *ProductType* on Level 2, and from the *OrderItem* objects on Level 0 to their concrete *Product* types. The object-type mixture on Level 1 (object *catStan*) is relatively static.

The context-aware multilevel rearchitecture in Fig. 13 gains in our priority factors of understandability and client visibility. In particular, preserving the *Product* class as an abstraction barrier above concrete *Product* classes in Level 1, removes the otherwise unavoidable duplication of methods, associations, code fragments and constraints (as in Fig. 4). Note that the pattern in Fig. 10 which is implemented in Fig. 11, requires special multilevel semantics to the association between the *Product* client class *Bundle* on Level 2, and the *ProductType* class. Our advice is to avoid coupling-based rearchitecture transformations like the pattern in Fig. 10, and follow factors that reflect the modeler's preferences.

[9] As in Fig. 11, the integrity constraint `quantityConstraint` on the *OrderItem* class is removed since it requires an MLM constraint language [11,16,61].

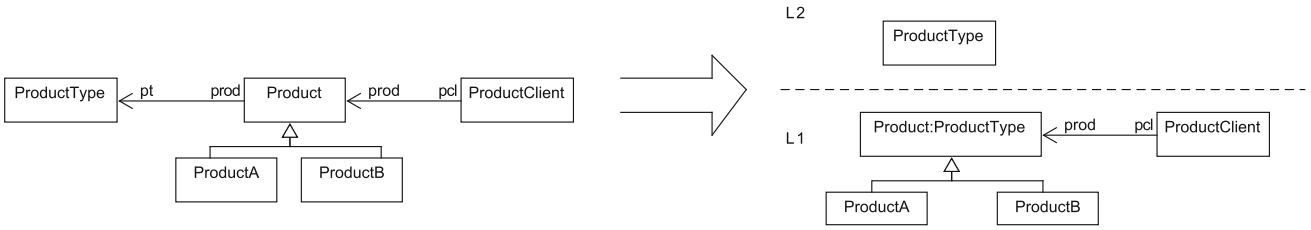



**Fig. 12** The MLM transformation for rearchitecting a class-hierarchy structure that has a client class

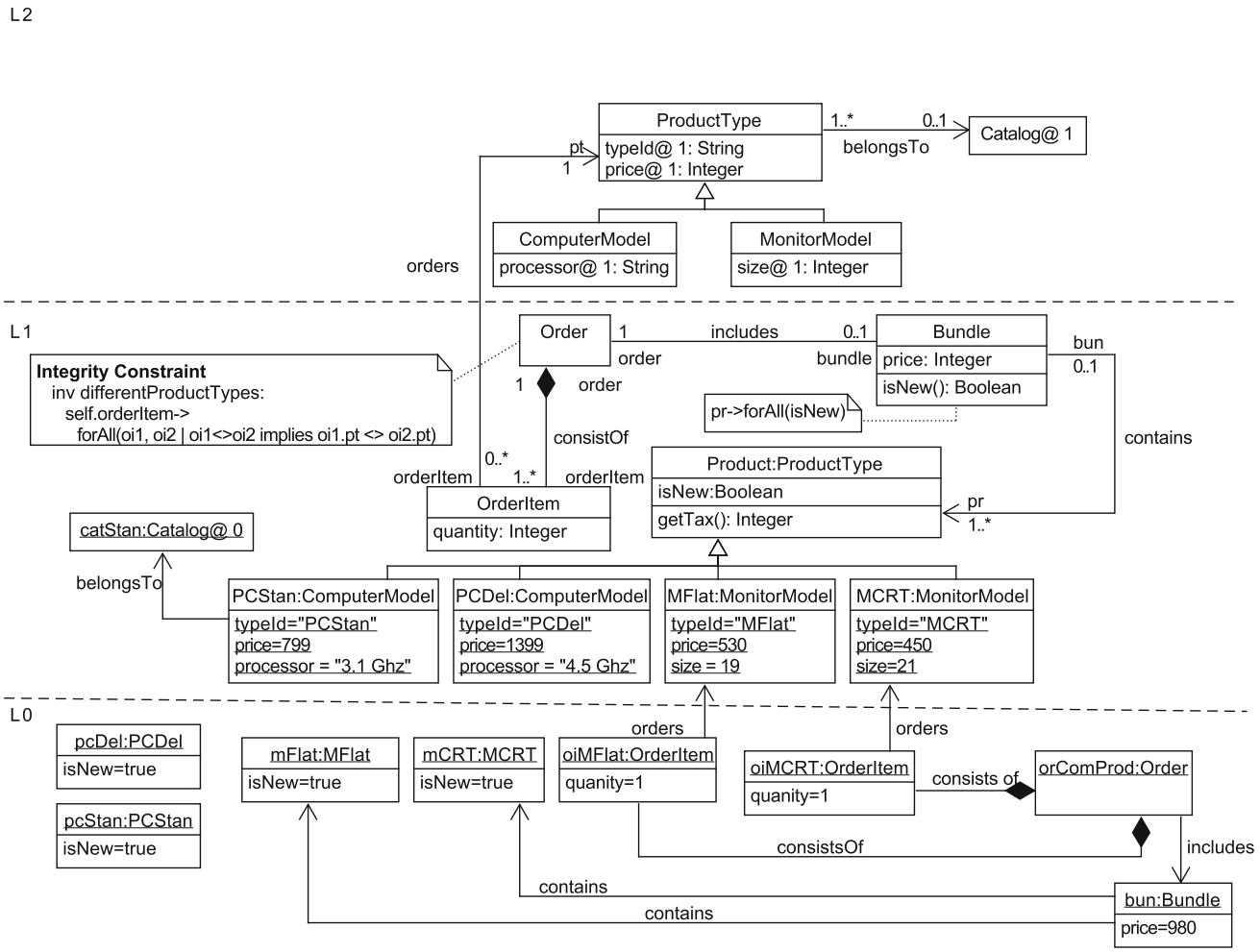



**Fig. 13** Context-aware multilevel rearchitecting of the *type-instance* structure in Fig. 2

**Table 3** Accidental complexity in Fig. 13

| Metric | Value |
|---|---|
| $\#intra-levelassociationsbetween$ | |
| *classes with different potency values* | $\#diffPot\_assoc(L1\_L2) = \frac{1}{4}$ |
| *#inter-level associations* | $\#intL\_assoc(L1\_L2) = \frac{1}{6}$ |
| *#inter-level links* | $\#intL\_lnk(L0\_L1\_L2) = \frac{1}{4}$ |
| *#object-clabject links in a level* | $\#lnk\_objTp(L1) = \frac{1}{4}$ |
| *Maximal potency difference in a level* | $pt\_diff(L2) = \frac{1}{2}$ |

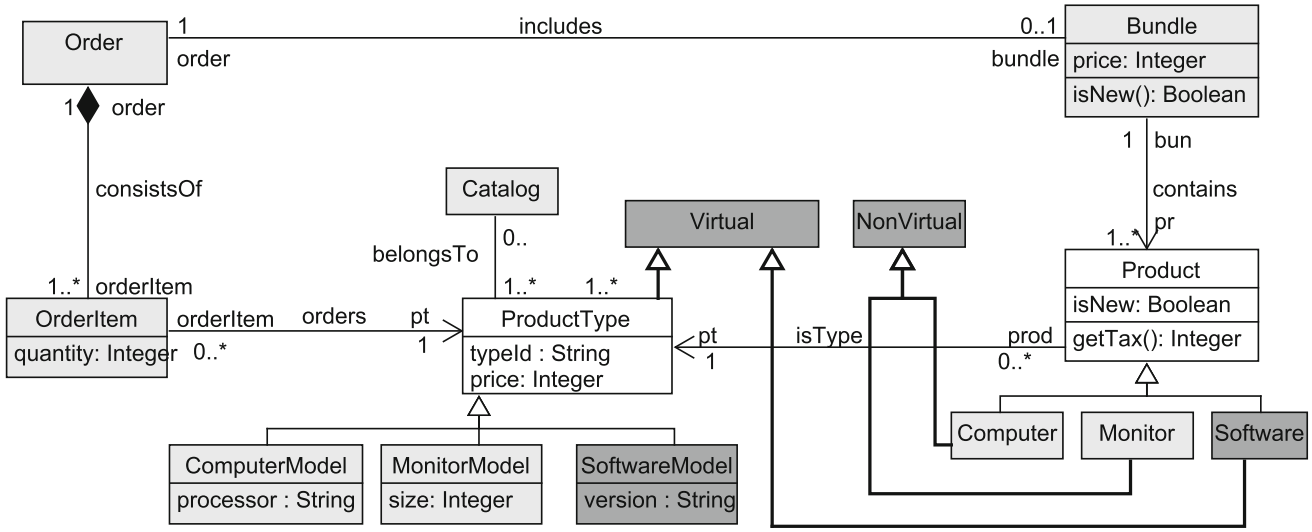



**Fig. 14** A class-hierarchy context added to the *type-instance* structure from Fig. 2. The new context classes are marked with dark grey background

## 5 MLM rearchitecture of models with *type-instance* and class-hierarchy structures

A class-hierarchy context for a *type-instance* structure refers to a hierarchy structure that includes one or both classes. Naturally, the tendency in rearchitecting is not to break class hierarchies between modeling levels, since it means major semantic changes, i.e., replacing subtyping by membership [11], and loss of visibility for client-classes. It is desirable to classify ancestors, descendants, and siblings on the same level. Figure 14 shows a complex class-hierarchy context for the *type-instance* structure from Fig. 2. The model is deliberately planned to present complex interaction of *type-instance* and hierarchy relationships.[10]

The natural advice for hierarchy context structures that involve exactly one of the *ProductType* or the *Product* classes is to put the hierarchy structure either on Level 2 or 1, respectively. Figure 15 shows a rearchitecture transformation that captures this advice. With respect to the *NonVirtual* superclass of classes *Computer* and *Monitor* in Fig. 14, it means that *NonVirtual* should be positioned on Level 1, together with its subclasses, as shown in Fig. 17.

Mixed hierarchy structures like the *Virtuality* structure in Fig. 14 present a *cohesion* modeling problem, since the superclass denotes a set that includes a type class *ProductType* and its member class *Software*. Apart from the cohesion problem, the classes of such a structure cannot be classified according to the previous advice. For example, in Fig. 17, *Virtual* should be on Level 2 (together with *ProductType*), while its subclass *Software* is on Level 1. But subclassing relationships cannot cross a level boundaries.

In complex *type-instance* hierarchy interaction, our advice, which is described in Fig. 16, is to replace the subclass relationship by a reference. Indeed, it changes the intended semantics, but it seems the right clean solution to the cohesion and to the MLM transformation problems. This transformation is applied in the MLM rearchitecture in Fig. 17.

### Proposed guidelines for MLM rearchitecture

The analyses of accidental-complexity factors in the above examples show that there is no single best transformation. The desired MLM transformation depends on one's values and ideals. The ideals that we adopt, ordered by their priority, are (1) understandability and conceptualization; (2) client visibility; (3) redundancy; and (4) level instability.

Following these ideals, we suggest the following transformation guidelines.[11] The guidelines identify interaction structures between a *type-instance* structure and its context classes and suggest a transformation, which is subject to the intended semantics of the context classes.

1. **Classes that are directly or indirectly associated with the *Instance* class, in a *type-instance* structure:** For each such class, determine whether its intended instantiation semantics is like that of *Instance*, i.e., a type of objects. In that case, put it on the level of the *Instance* class and preserve the association structure. Classes *OrderItem, Order*, and *Bundle* in Fig. 13 are such classes. If there is an association cycle between the *Instance* and the *Type* classes, this advice creates an inter-level association.

[10] For simplicity, method text and constraints are removed.

[11] This is not a refactoring since replacing subtyping or association by type membership changes the semantics.

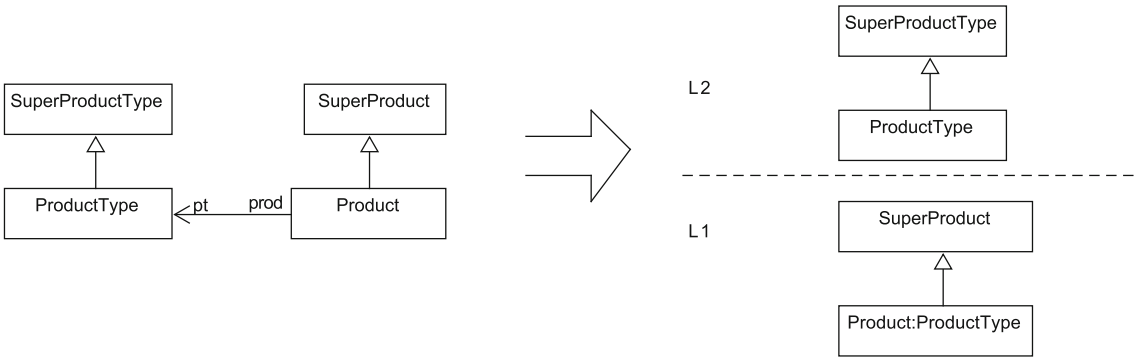



**Fig. 15** The MLM transformation for rearchitecting a simple class hierarchy structure of the *type-instance* relationship

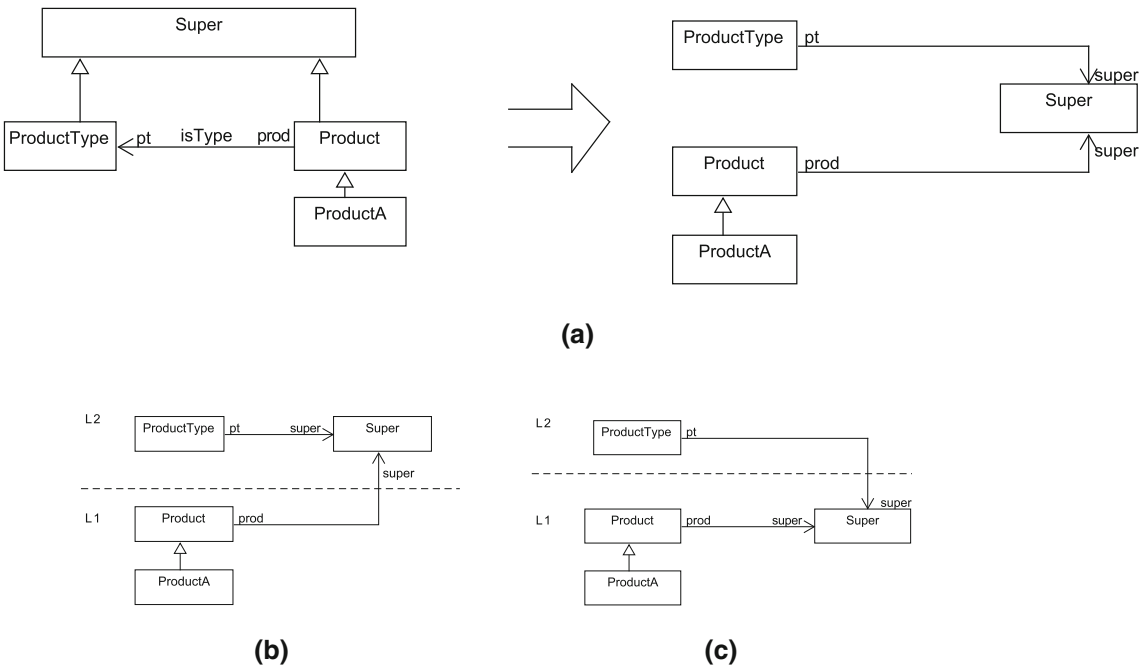



**Fig. 16** The MLM transformation for rearchitecting a mixed class hierarchy structure of the *type-instance* relationship

2. **Classes that are directly or indirectly associated with the *Type* class, in a *type-instance* structure:** Such classes, like *Catalog* in Fig. 14, join the *Type* class, in the same level (see Fig. 17), provided that: (1) they are not already positioned in a different level, following the advice of previous entry; (2) their intended instantiation semantics is like that of *Type*, i.e., a type of types or a type of virtual objects. Otherwise, they are positioned according to their intended semantics.
3. **Sibling (through hierarchy) classes of transformed classes:** Sibling classes would better reside on the same level in the MLM architecture (to enable client visibility of hierarchy classes). For example, for Fig. 13, the advice of this step is to classify sibling classes of say, *Order* or *Bundle* together, on the same level.
4. **Ancestor or descendant classes of transformed classes:** In principle, ancestor and descendant classes should go together. In cases of deep class-hierarchy structures, or if hierarchy structures include classes that are already classified on different levels, more heuristic advice is needed, to balance accidental-complexity factors with modeling ideals.

## 6 Type-instance contexts in real models

In previous sections, we have studied factors of MLM accidental complexity, suggested metrics for their quantitative evaluation, and ended with general guidelines for MLM rearchitecture of standard two-level models. The MLM complexity factors emphasize the major role of inter-level

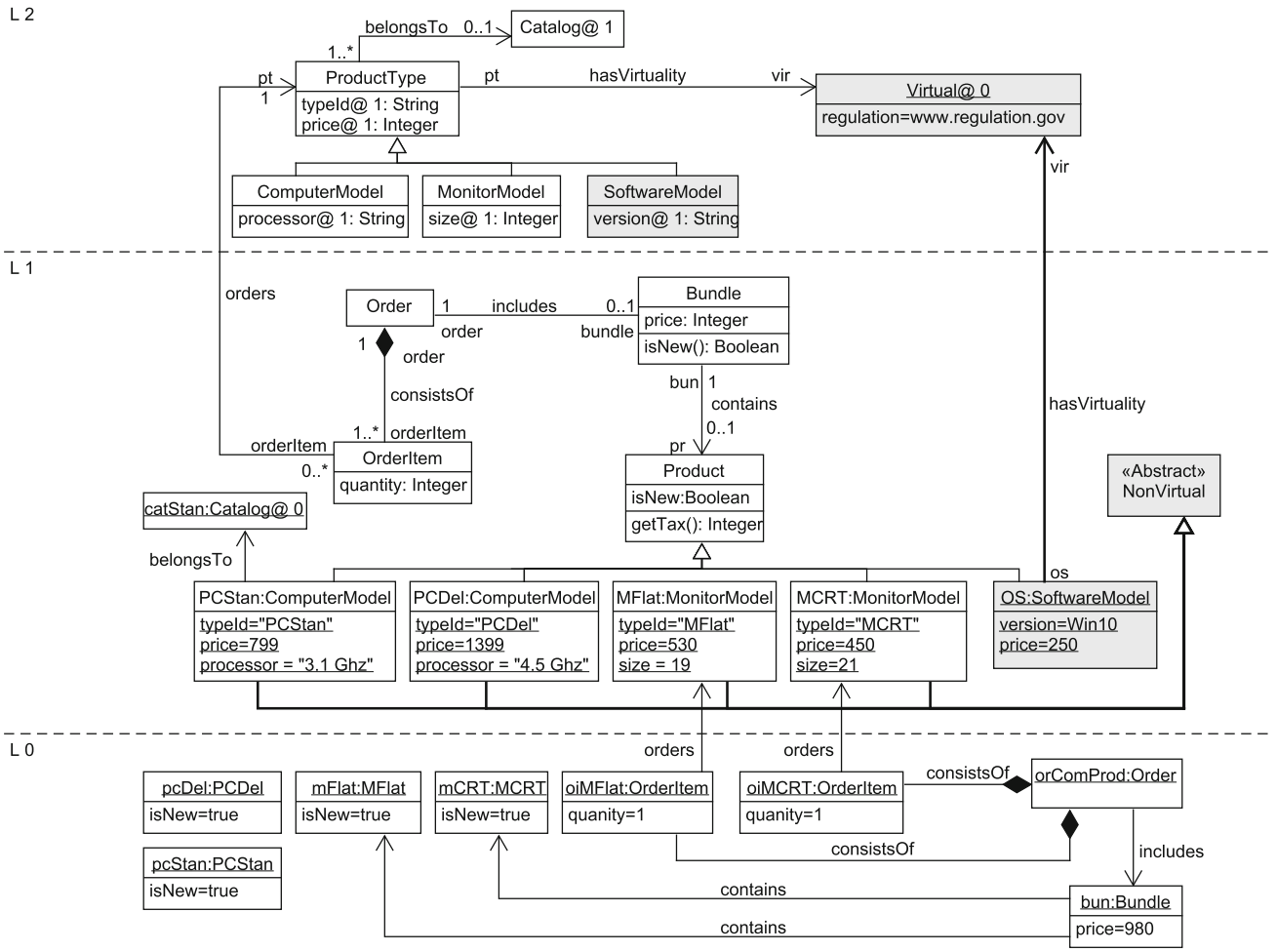



**Fig. 17** Context-aware multilevel rearchitecting of the *type-instance* and class-hierarchy structure in Fig. 14

interaction in determining MLM complexity. A model with little inter-level interaction has reduced accidental complexity. Yet, as shown by the running example of the paper, when a *type-instance* structure is embedded within a complex context that involves an association cycle and/or a complex class-hierarchy structure, inter-level interaction is unavoidable. An MLM rearchitecture of such models must admit some amount of inter-level interaction. The role of modelers is to find a desired balance of MLM decisions, so to minimize accidental complexity according to their modeling ideals.

In this section, we check whether complex MLM contexts arise in real-world models. Otherwise, if all (or most) real-world two-level models are "well structured," i.e., the context classes of the *Type* class in a *type-instance* structure are not related to those of the *Instance* class, then MLM modeling is rather simple, and most accidental complexity factors do not arise. Figure 18 sketches four structures of complex *type-instance* contexts that enforce inter-level MLM interaction. First, we investigate *type-instance* contexts in real-world models and then study their occurrence in a model from the Umple repository [62] and suggest an MLM rearchitecture, following the guidelines from previous section.

## 6.1 Investigation of *type-instance* contexts in Real-World Models

Our study of the relevance of the context-aware MLM accidental-complexity factors consists of investigation of contexts in which *type-instance* structures occur in real models (and meta-models) in different domains. The models were collected from a variety of sources, including the data provided in [5],[12] the ATL meta-model Zoo [63], the Umple repository [62], and the models in [64–66].

Table 4 summarizes the models under investigation, their domains, their size in terms of number of classes (c), associations (r)[13], and identified *type-instance* occurrences. A large number of identified occurrences is marked as $> 5$.

[12] This paper points to a site that lists *type-instance* occurrences in meta-models, http://miso.es/multilevel/multi-level-patterns.htm.

[13] Sizes of ATL models are taken from [5].

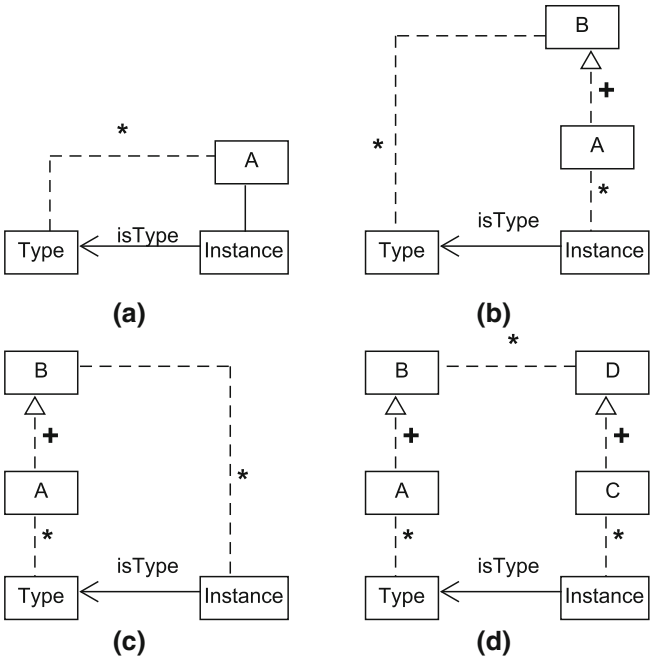



**Fig. 18** Four structures of complex *type-instance* contexts

A dashed line denotes zero or more association or class-hierarchy relations; a $*$ label denotes zero or more relations and a $+$ label denotes at least one relation

**Table 4** Type-instance context occurrences in selected meta-models

| Name | Domain | Size | number of occurrences |
|---|---|---|---|
| ANT [63] | Enterprise/Process Model | 48c, 28r | $> 5$ |
| ACME [63] | Software Architecture | 16c, 13, r | $> 5$ |
| ProMARTE [63] | Software Architecture | 180c, 142r | 2 |
| Agate [63] | Enterprise/Process Model | 69c, 123r | $> 5$ |
| SWRC 1 [63] | Bibliographic data | 55c, 68r | $> 5$ |
| SCADE [63] | Systems/Software | 106c, 231r | $> 5$ |
| Cobol Meta model [63] | Programming language | 13c, 22r | $> 5$ |
| ifc2x3 [63] | Information Modeling Language | 699c, 592r | $> 5$ |
| Umple [62] | Manufacturing Plant Controller | 11c, 5r | 2 |
| Umple [62] | Organization Decision making | 14c, 12r | $> 5$ |
| Business Activity RBAC [66] | Business Activity | 6c, 13r | $> 5$ |
| Open Cloud Computing Interface [65] | Software Architecture Domain | 25c, 34r | $> 5$ |
| ArchiMeDeS [64] | Software Architecture | 25c, 34r | $> 5$ |

We have analyzed these models and found that most models include *type-instance* occurrences that are embedded in complex contexts, following the structures in Fig. 18. This analysis supports our claim that context-aware accidental complexity has a central role in MLM.

Below, we list some such small contexts. A context is presented as a sequence of classes, where consecutive classes are related by an association (denoted by a line) or by subclass ($\preceq$) or a superclass ($\succeq$) relation:

1. **The ANT model** [63]: Context paths for type class `TaskDef` and instance class `NewTask` demonstrate context structures (b) and (d),

(a) $\texttt{TaskDef} \xrightarrow{\textbf{isTypeOf}} \texttt{NewTask} \preceq \texttt{Task}$—`Target`—`Project`—`TaskDef`;

(b) $\texttt{TaskDef} \xrightarrow{\textbf{isTypeOf}} \texttt{NewTask} \preceq \texttt{Task}$—`Target`—`Java`—`Project`—`TaskDef`;

(c) $\texttt{TaskDef} \xrightarrow{\textbf{isTypeOf}} \texttt{NewTask} \preceq \texttt{Task} \succeq \texttt{PredefinedTask} \succeq \texttt{ClassPath}$— `FileSet`—`Path`—`Project`—`TaskDef`;

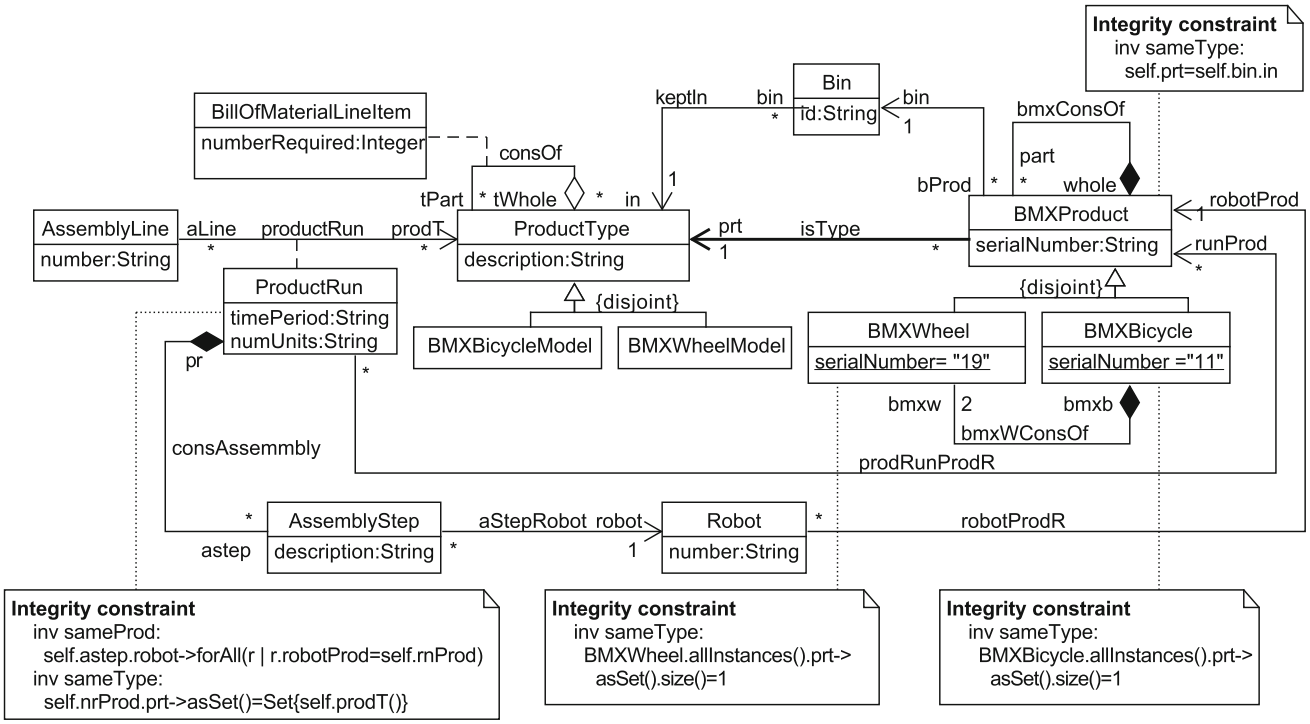



**Fig. 19** A model of the Manufacturing Plant Controller in [62]

2. **The ProMARTE model** [63]: Context path for type class `Classifier` and instance class `Instance Specification` demonstrates context structure d. `Classifier` **isTypeOf** `InstanceSpecific ation` $\preceq$ `AnnotatedModelElement` $\succeq$ `Classifier`;
3. **The SCADE** [63]**model:** Context path for type class `AnnNoteType` and instance class `AnnNote` demonstrates context structure a. `AnnNoteType` **isTypeOf** `AnnNote—AnnAttValue—AnnAttDefinition— AnnAttGroup—Ann NoteType`;
4. **The Manufacturing Plant Controller** [62] model: Context path for type class `ProductType` and instance class `Product` demonstrates context structure a. `ProductType` **isTypeOf** `BMXProduct—ProductRun—ProductType`;
5. **The Organization Decision Making** [62] **model:** Context path for type class `DecisionType` and instance class `Decision`; demonstrates context structure a. `DecisionType` **isTypeOf** `Decision—DecisionByBody—DecisionMakingBody —ApprovalLevel—DecisionType`;
6. **The Open Cloud Computing Interface** [65] **model:** Context path for type class `Kind` and instance class `Entity` demonstrates context structure a. `Kind` **isTypeOf** `Entity—Mixing—Action—Kind`;
7. **The Business Activity RBAC**[66] **model:** Context path for type class `TaskType` and instance class `TaskInstance` demonstrates context structure a. `TaskType` **isTypeOf** `TaskInstance—Subject—Role—TaskType`;

## 6.2 *Type-instance* and MLM Rearchitecture of a two-level model from Umple [67–69]

The Umple repository [62] presents a model of a Manufacturing Plant Controller that constructs several types of mechanical devices. It has assembly lines that can be used, each, in the manufacturing of any of its products. Figure 19 presents a simplified class model of the plant. The class model includes a *type-instance* structure within a complex context of association cycles.[14]

[14] For the sake of visual simplicity, the two complex constraints on class *BMXProduct* are omitted: (1) if $pr \in bmxPr.part$, then $pr.prt \in bmxPr.prt.tPart$; (2) for every set $Prts$ of parts of a *BMXProduct* object $bmxPr$, such that $Prts$ objects share a common *productType* $prtTp$, the size of $Prts$ is given by the *requiredNumber* attribute of the *BillOfMaterialLineItem* of the link $\langle bmxPr.prt, prtTp\rangle$; i.e., $|Prts| = \langle bmxPr.prt, prtTp\rangle.BillOfMaterialLineItem.requiredNumber$. Proper formulation of this constraint requires definition of auxiliary functions within a local computation scope

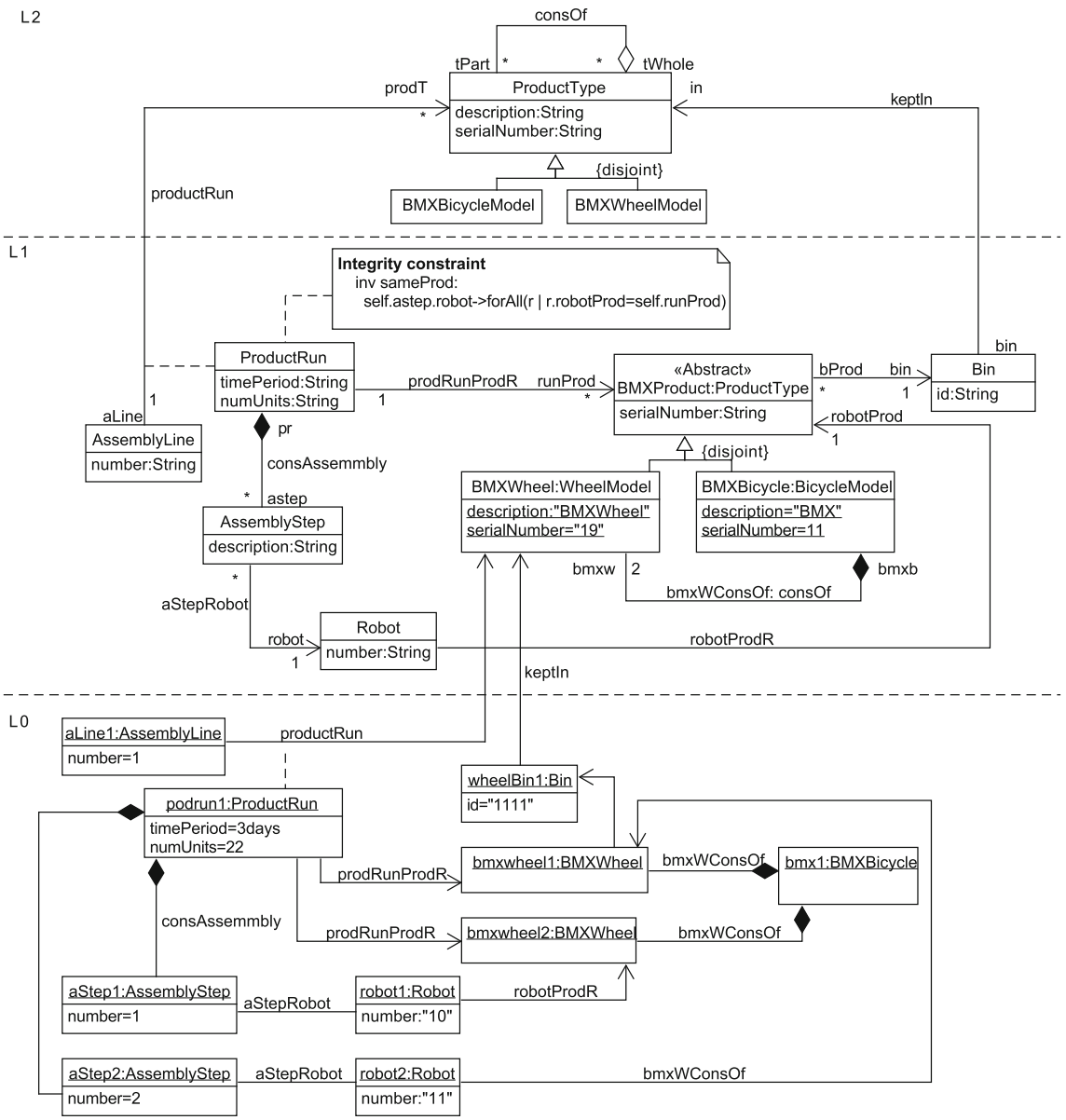



**Fig. 20** MLM rearchitecting of the class model in Fig. 19

Figure 20 presents the result of applying the guidelines for MLM transformation from Sect. 5 to the class model in Fig. 19, extended with a partial initial state in Level 0. The guideline preferences put the *conceptualization* and *understandability* factors (with preservation of *client visibility*) in the first place, followed by *redundancy* and then *stability*.

1. ***Instance* related classes:** Class *BMXProduct* is positioned on Level 1. Classes *ProductRun, AssemblyStep, Robot*, and *Bin* that are directly or indirectly associated with *BMXProduct* are positioned in Level 1 because they describe concrete objects in the plant, i.e., they are types of objects.
2. ***Type* related classes:** Class *ProductType* is positioned on level 2, with potency 2. Classes *BillOfMaterialLineItem* and *AssemblyLine* are directly associated with *ProductType* and are not already positioned on Level 1. *BillOfMaterialLineItem* becomes redundant in MLM since its intention is to restrict the size of *tpart-twhole* aggregations between products, based on their types. Size restrictions on associations in class models are specified by multiplicities. Therefore, *BillOfMaterialLineItem* is removed, and the *bmxWconsOf* association on Level 1 species the required number of parts for its *bmxw* property. *AssemblyLine* describes concrete objects in a plant, i.e., represents a type of data objects on Level 0. There-

**Table 5** Accidental complexity in the MLM rearchitecture of the model from Umple in Fig. 20

| Metric | Value |
|---|---|
| $\#inter-levelassociations$ | $\#intL\_assoc(L1\_L2) = \frac{2}{9}$ |
| $\#inter-levellinks$ | $\#intL\_lnk(L0\_L1\_L2) = \frac{2}{12}$ |

fore, it is positioned in Level 1, implying an inter-level association between *AssemblyLine* and *ProductType*.

3. **Sibling classes:** None.
4. **Ancestor and Subclasses:** Classes *BMXBicycle*, and *BMXWheel*, the *Product* subclasses are positioned on Level 1. Class *BMXProduct* is not removed, because there are client classes, and there is a need to preserve their visibility, for example, references and operations of *ProductRun* and *Robot* objects. Classes *BMXBicycleModel* and *BMXWheelModel*, the subclasses of *ProductType*, are positioned on level 2, with potency 2, because they describe types of types. (Their instances are virtual.)

**Constraints in the multilevel model** Constraint *sameProd* of class *ProductRun*, being an intra-level constraint, stays as is. Constraint *sameType* of class BMXProduct requires an MLM constraint language and is not listed. Interestingly, the MLM structure already accounts for the two *sameType* constraints of the *BMXProduct* subclasses, since the built-in memberships of *BMXBicycle* and *BMXWheel* guarantee that their objects have the same product type. Therefore, these constraints are removed. Similarly, the complex constraints on *BMXProduct* are redundant, since the required number of parts is specified by the multiplicity constraints of the *whole-part* associations between *BMXProduct* subclasses.

The multilevel model admits some inter-level associations and links, but all are directed from a level towards its adjacent higher one. Therefore, in terms of instability, it is always a less stable (more dynamically changing) level that depends on a more stable one (downward level coupling). The levels are cohesive (no mixture of linked objects and classes), and there is no redundancy. Altogether, there is only a level-instability accidental complexity, measured by the metrics in Table 5.

## 7 Conclusion and future work

In this paper, we focused on contexts of *type-instance* structures, and their role in determining the accidental complexity of a multilevel model. We identified multiple factors of accidental complexity in MLM and showed that they arise in structures with complex context. We provided quantitative metrics for these factors and used them to compare accidental complexity of MLM transformations, along different parameters of a multilevel model.

We have shown that there are complex MLM contexts in which no silver bullet can remove all accidental-complexity factors. Therefore, modelers need their subjective MLM scales for balancing conflicting accidental-complexity factors. We suggested guidelines for MLM rearchitecture transformation that maximizes *understandability* and *client visibility*.

The relevance of this context-aware analysis of MLM accidental complexity is validated by an experimental study of *type-instance* occurrences in multiple real-world models. We found that most models include complex contexts with *type-instance* occurrence. Finally, we have demonstrated the application of the suggested MLM rearchitecture transformation to a real model that is taken from the UMPLE site.

In the future, we plan a user study of the *conceptualization* and *understandability* factors in MLM. We think that experiments in the design, understanding, and application of MLM models are needed in order to clarify their useful aspects. Such experiments will also help in the practical design and evaluation of accidental complexity metrics. In addition, we continue to develop the MLM component of our FOML tool [16] and plan on using it in an industrial project we are involved in.

## References

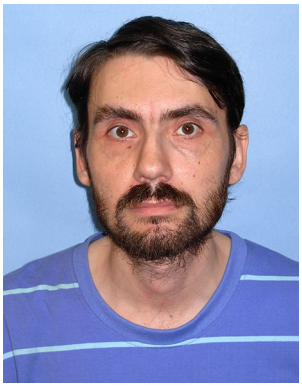
1. Henderson-Sellers, B.: On the Mathematics of Modelling, Metamodelling, Ontologies and Modelling Languages. Springer Science & Business Media, Berlin (2012)
2. Brooks, F.P.: No silver bullet—essence and accident in software engineering. IEEE Computer **20**, 10–19 (1987)
3. Kuehne, T., Schreiber, D.: Can programming be liberated from the two-level style: multi-level programming with deepjava. ACM SIGPLAN Notes **42**, 229–244 (2007)
4. Atkinson, C., Kühne, T.: Reducing accidental complexity in domain models. Softw. Syst. Model. **7**, 345–359 (2008)
5. de Lara, J., Guerra, E., Cuadrado, J.S.: When and how to use multilevel modelling. ACM Trans. Softw. Eng. Methodol. **24** (2014) 12: 1–12:46
6. de Lara, J., Guerra, E., Cuadrado, J.S.: Model-driven engineering with domain-specific meta-modelling languages. SoSyM **14**, 429–459 (2013)
7. de Lara, J., Guerra, E., Cobos, R., Moreno-Llorena, J.: Extending deep meta-modelling for practical mde. Comput. J. **57**, 36–58 (2014)
8. Atkinson, C., Kühne, T.: The essence of multilevel metamodeling. In: UML. Springer (2001) 19–33
9. Atkinson, C., Kühne, T.: Rearchitecting the uml infrastructure. ACM Trans. Model. Comput. Simul. **12**, 290–321 (2002)
10. Rossini, A., de Lara, J., Guerra, E., Rutle, A., Lamo, Y.: A graph transformation-based semantics for deep metamodelling. In: Applications of Graph Transformations with Industrial Relevance. Springer (2012)

11. Balaban, M., Khitron, I., Kifer, M., Maraee, A.: Formal executable theory of multilevel modeling. In: CAISE. (2018)
12. Balaban, M., Khitron, I., Kifer, M., Maraee, A.: Multilevel modeling: what's in a level? a position paper. In: MODELS Workshops, MULTI-2018. (2018)
13. Rushton, A., Croucher, P., Baker, P.: The handbook of logistics and distribution management: Understanding the supply chain. Kogan Page Publishers (2014)
14. Scholz-Reiter, B., Sowade, S., Hildebrandt, T., Rippel, D.: Production management modeling of orders in autonomously controlled logistic systems. Prod. Eng. **4**, 319–325 (2010)
15. Litium docs: Litium Domain model of ERP connect API order contracts. https://bit.ly/3p8ER7u (2020) Accessed: Jan 31, 2020
16. Khitron, I., Balaban, M., Kifer, M.: FOML Site. https://goo.gl/AgxmMc (2021)
17. Fowler, M.: Analysis Patterns: Reusable Object Models. Addison-Wesley, Boston (1997)
18. Johnson, R., Woolf, B.: The Type Object Pattern. http://web.comlab.ox.ac.uk/oucl/work/jeremy.gibbons/dpa/typeobject.pdf (1996)
19. Nystrom, R.: Game Programming Patterns. Genever Benning (2014)
20. Cardelli, L.: Structural subtyping and the notion of power type. In: Proceedings of the 15th ACM SIGPLAN-SIGACT symposium on Principles of programming languages. (1988) 70–79
21. Odell, J.: Power types. J. Object Oriented Program. **7**, 8–12 (1994)
22. Gonzalez-Perez, C., Henderson-Sellers, B.: A powertype-based metamodelling framework. Softw. Syst. Model. **5**, 72–90 (2006)
23. Object Management Group (OMG): OMG Unified Modeling Language. Specification Version 2.5, OMG (2015)
24. Carvalho, V.A., Almeida, J.P.A.: Toward a well-founded theory for multi-level conceptual modeling. Softw. Syst. Model. **17**, 205–231 (2016)
25. Almeida, J., Fonseca, C., Carvalho, V.: A comprehensive formal theory for multi-level conceptual modeling. In: ER. (2017) 280–294
26. Jeusfeld, M.A., Neumayr, B.: Deeptelos: Multi-level modeling with most general instances. ER **2016**, 198–211 (2016)
27. Jeusfeld, M.A., Almeida, J.P.A., Carvalho, V.A., Fonseca, C.M., Neumayr, B.: Deductive reconstruction of mlt* for multi-level modeling. In: MODELS Workshops, MULTI-2020. (2020)
28. Jarke, M., Gallersdörfer, R., Jeusfeld, M.A., Staudt, M.: Conceptbase—A deductive object base for meta data management. J. Intell. Inf. Syst. **4**, 167–192 (1995)
29. Gogolla, M., Sedlmeier, M., Hamann, L., Hilken, F.: On metamodel superstructures employing UML generalization features. In: MULTI'2014. (2014)
30. Gogolla, M., Bohling, J., Richters, M.: Validating UML and OCL Models in USE by Automatic Snapshot Generation. J. Softw. Syst. Model. **4**, 386–398 (2005)
31. Kuhlmann, S.M., Hamann, L., Gogolla, M.: Extensive Validation of OCL Models by Integrating SAT Solving into USE. In: TOOLS EUROPE 2011. Volume 6705. (2011) 290–306
32. Almeida, J., Musso, F., Carvalho, V., Fonseca, C., Guizzardi, G.: Preserving multi-level semantics in conventional two-level modeling techniques. In: ER. (2019) 142–151
33. Hinkel, G.: Using structural decomposition and refinements for deep modeling of software architectures. Softw. Syst. Model. **18**, 2787–2819 (2019)
34. Kühne, T.: Exploring potency. In: Proceedings of the 21th ACM/IEEE International Conference on Model Driven Engineering Languages and Systems. (2018) 2–12
35. Lange, A., Atkinson, C.: On the rules for inheritance in lml. In: 2019 ACM/IEEE 22nd International Conference on Model Driven Engineering Languages and Systems Companion (MODELS-C). (2019) 113–118
36. Balaban, M., Kifer, M.: Logic-Based Model-Level Software Development with F-OML. In: MoDELS 2011. (2011)
37. Balaban, M., Khitron, I., Kifer, M.: Logic-based software modeling with FOML. J. Object Technol. **19**(3), 1–21 (2020)
38. de Lara, J., Guerra, E.: Deep Meta-modelling with METADEPTH. In: the 48th International Conference on Objects, Models, Components, Patterns. (2010)
39. Demuth, A., Lopez-Herrejon, R.E., Egyed, A.: Cross-layer modeler: a tool for flexible mlm with consistency checking. In: ACM SIGSOFT symp. (2011)
40. Lange, A., Atkinson, C.: Multi-level modeling with melanee. In: MULTI 2018 Workshop co-located with MODELS 2018. (2018) 653–662
41. Frank, U.: Multilevel modeling- toward a new paradigm of conceptual modeling and information systems design. Bus. Inf. Syst. Eng. **6**, 319–337 (2014)
42. de Kinderen, S., Kaczmarek-Heß, M.: On model-based analysis of organizational structures: an assessment of current modeling approaches and application of multi-level modeling in support of design and analysis of organizational structures. Software and Systems Modeling (2019)
43. Mylopoulos, J., Borgida, A., Jarke, M., Koubarakis, M.: Telos: representing knowledge about information systems. ACM TOIS **8**, 325–362 (1990)
44. Kifer, M., G., L., Wu, J.: Logical foundations of object-oriented and frame-based languages. Journal of the ACM **42** (1995) 741–843
45. Neumayr, B., Jeusfeld, M., Schrefl, M., Schütz, C.: Dual deep instantiation and its conceptbase implementation. In: Intl. Conf. on Advanced Information Systems Eng. (2014) 503–517
46. Igamberdiev, S.M., Grossmann, G., Stumptner, M.: An Implementation of Multi-Level Modelling in F-Logic. In: 1st International Workshop on Multi-Level Modeling (Multi 2014). (2014)
47. Neumayr, B., Schuetz, C.G., Jeusfeld, M.A., Schrefl, M.: Dual deep modeling: MLM with dual potencies and its formalization in F-Logic. SoSyM (2016)
48. Igamberdiev, M., Grossmann, G., Selway, M., Stumptner, M.: An integrated multi-level modeling approach for industrial-scale data interoperability. Software & Systems Modeling , 269–294 (2018)
49. Haslum, P., et al.: Reducing accidental complexity in planning problems. IJCA **I**, 1898–1903 (2007)
50. Gerbig, R.: Deep, seamless, multi-format, multi-notation definition and use of domain-specific languages. PhD thesis, School of Business Informatics and Mathematics, University of Mannheim (2017)
51. Kuhne, T., Lange, A.: Meaningful metrics for multi-level modelling. In: MODELS Workshops, MULTI-2020. (2020)
52. Balaban, M., Khitron, I., Maraee, A.: Context-aware factors in rearchitecting two-level models into multilevel models. In: MODELS Workshops, MULTI-2018. (2018)
53. Fowler, M., Beck, K.: Refactoring: improving the design of existing code. Addison-Wesley Professional, Boston (1999)
54. Btiand, L., J., W., Lounis, H., Ikomomvski, S.: A Comprehensive Investigation of Quality Factors in Object-oriented Designs: An Industrial Case Study. In: The 21st International Conference on Software Engineering. (1999) 345–354
55. Genero, M., Piatini, M., Manso: Finding early indicators of uml class diagrams understandability and modifiability. In: Proceedings. 2004 International Symposium on Empirical Software Engineering, 2004. ISESE '04. (2004) 207–216
56. Manso, M.E., Genero, M., Mario, P.: No-redundant metrics for uml class diagram structural complexity. In: International Conference on Advanced Information Systems Engineering. (2003) 127–142
57. Genero, M., Manso, E., Visaggio, A., Canfora, G., Piattini, M.: Building measure-based prediction models for uml class diagram maintainability. Empirical Softw. Eng. **12**, 517–549 (2007)

58. Bagheri, E., Gasevic, D.: Assessing the maintainability of software product line feature models using structural metrics. Softw. Quality J. **19**, 579–612 (2011)
59. Cruz-Lemus, J.A., Maes, A., Genero, M., Poels, G., Piattini, M.: The impact of structural complexity on the understandability of uml statechart diagrams. Inf. Sci. **180**, 2209–2220 (2010)
60. Acherkan, E., Hen-Tov, A., Lorenz, D.H., Schachter, L.: The ink language meta-metamodel for adaptive object-model frameworks. In: ACM Int. Conf. Comp. on OO Prog. Sys. Lang. and Appl. Companion. OOPSLA, 181–182 (2011)
61. Atkinson, C., Gerbig, R., Kuhne, T.: Opportunities and Challenges for Deep Constraint Languages. In: 15th International Workshop on OCL and Textual Modeling. (2015)
62. Umple Online Examples. https://github.com/umple/Umple/wiki/examples (2016) Accessed: Jun 29, 2016
63. The Metamodel Zoos. https://web.imt-atlantique.fr/x-info/atlanmod/index.php?title=Zoos (2015) Accessed: 9 July 2015
64. López-Sanz, M., Marcos, E.: Archimedes: a model-driven framework for the specification of service-oriented architectures. Inf. Syst. **37**, 257–268 (2012)
65. Merle, P., Barais, O., Parpaillon, J., Plouzeau, N., Tata, S.: A precise metamodel for open cloud computing interface. In: 2015 IEEE 8th International Conference on Cloud Computing, IEEE (2015) 852–859
66. Strembeck, M., Mendling, J.: Modeling process-related rbac models with extended uml activity models. Inf. Softw. Technol. **53**, 456–483 (2011)
67. Lethbridge, T., Forward, A., Badreddin, O., Brestovansky, D., Garzon, M., Aljamaan, H., Eid, S., Orabi, A., Orabi, M., Abdelzad, V.: Umple: Model-driven development for open source and education. Science of Computer Programming **208**,(2021)
68. Lethbridge, T.C., Algablan, A.: Umple: an executable uml-based technology for agile model-driven development. In: Advancements in Model-Driven Architecture in Software Engineering. IGI Global (2021) 1–25
69. UmpleOnline. http://cruise.eecs.uottawa.ca/umpleonline/ (2021) Accessed: January, 2021




**Mira Balaban** is a Professor Emerita in the Computer Science Department at Ben Gurion University, Israel. She is also a graduate of music performance from the Rubin Academy of Music in Tel-Aviv. Her research interests include software modeling, with emphasis on correctness, optimization, languages and inference of models, programming languages, and Computer Music.



**Igal Khitron** is a Ph.D. student in the Department of Computer Science at the Ben Gurion University in Israel. His main research concerns the development of the FOML software modeling language, and its FOModeLer application.

**Azzam Maraee** is a faculty member within the Department of Management Information Systems, Achva Academic College and adjunct lecturer at the Department of Computer Science, Ben-Gurion University of the Negev. He has B.Sc. in mathematics, M.Sc. in information system engineering, and Ph.D. in computer science all from Ben-Gurion University, Israel. His research focuses on software engineering, with emphasis on modeling: model correctness and reasoning, modeling languages, and model patterns.