

Refactoring multi-level models

JUAN DE LARA, Universidad Autónoma de Madrid (Spain)

ESTHER GUERRA, Universidad Autónoma de Madrid (Spain)

Multi-level modelling promotes flexibility in modelling by enabling the use of several meta-levels instead of just two, as is the case in mainstream two-level modelling approaches. While this approach leads to simpler models for some scenarios, it introduces an additional degree of freedom as designers can decide the meta-level where an element should reside, having to ascertain the suitability of such decisions.

In this respect, model refactorings have been successfully applied in the context of two level-modelling to rearrange the elements of a model while preserving its meaning. Following this idea, we propose a catalogue of 17 novel refactorings specific to multi-level models. Their objective is to help designers in rearranging elements across and within meta-levels and exploring the consequences. In this paper, we detail each refactoring in the catalogue, show a classification across different dimensions, and describe the support we provide in our METADEPTH tool. We present two experiments to assess two aspects of our refactorings. The first one validates the predicted semantic side effects of the refactorings on the basis of more than 210.000 refactoring applications. The second one measures the impact of refactorings on three quality attributes of multi-level models.

Categories and Subject Descriptors: [Software and its engineering]: Model-driven software engineering; Domain specific languages; Design languages; Software design engineering

Additional Key Words and Phrases: Meta-modelling, Multi-level modelling, Model refactoring, METADEPTH

ACM Reference Format:

Juan de Lara and Esther Guerra, 2018. Refactoring multi-level models. *ACM Trans. Softw. Eng. Methodol.* V, N, Article A (January YYYY), 55 pages.

DOI : <http://dx.doi.org/10.1145/0000000.0000000>

1. INTRODUCTION

Model-Driven Engineering (MDE) [Schmidt 2006; Brambilla et al. 2012] promotes models as the main assets of the software development process. Hence, models are used to describe, understand, reason, simulate, test and generate code for the final system, among other activities. The set of valid models that can be constructed is described by a meta-model, which declares the admissible model types, properties and relations, as well as the constraints that models should fulfil. Then, models are built by instantiating this meta-model. Hence, MDE has traditionally adopted a *two-level, top-down* meta-modelling approach, where meta-models reside in a certain meta-level, and models are created one meta-level below by using types from the meta-model [de Lara and Guerra 2017].

In contrast, *multi-level* modelling approaches allow organizing models and their descriptions in several meta-levels, not necessarily two. This has been shown to reduce the accidental complexity of models in certain scenarios, typically, when the type-object

Author's addresses: J. de Lara, Computer Science Department, Universidad Autónoma de Madrid (Spain); E. Guerra, Computer Science Department, Universidad Autónoma de Madrid (Spain).

Permission to make digital or hard copies of part or all of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies show this notice on the first page or initial screen of a display along with the full citation. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, to republish, to post on servers, to redistribute to lists, or to use any component of this work in other works requires prior specific permission and/or a fee. Permissions may be requested from Publications Dept., ACM, Inc., 2 Penn Plaza, Suite 701, New York, NY 10121-0701 USA, fax +1 (212) 869-0481, or permissions@acm.org.

© YYYY ACM 1049-331X/YYYY/01-ARTA \$15.00

DOI : <http://dx.doi.org/10.1145/0000000.0000000>

pattern or one of its variants arise [Atkinson and Kühne 2008; de Lara et al. 2014]. In this way, multi-level modelling enhances flexibility in MDE by providing modellers with the freedom to design how the different model elements should be organized in levels. However, this freedom may also become a burden, as modellers may need to try different design alternatives in order to find an optimal organization of model elements in meta-levels, or they may have to reconsider previous decisions and subsequently repair a possible large set of instances at several meta-levels below. Performing these rearrangements by hand is costly, time consuming and error prone.

In the context of programming, refactoring [Fowler 1999; Opdyke and Johnson 1990] is a technique to restructure a piece of code without changing its external behaviour. It can be used for different purposes, like reducing complexity, enhancing comprehensibility, increasing efficiency or reducing memory footprint. Refactorings have also been adapted to work with models [Mens and Tourwé 2004] – most notably for UML [Sunyé et al. 2001] – ontologies [Gröner et al. 2010], process models [Weber et al. 2011; Fdhila et al. 2015], architectures [Zimmermann 2015] and databases [Vial 2015]. However, refactorings in a multi-level setting are currently lacking. These would need to consider not only the rearrangement of elements within a model at a given meta-level, but also the possibility to move up or down information across levels, creating new meta-levels, compacting existing ones, or repairing existing instances at lower meta-levels.

In this paper, we provide a classification of multi-level refactorings, and present a catalogue with the definition of 17 novel refactorings including their pre-conditions and consequences. Multi-level refactorings capture common reorganizations of multi-level models. Just like standard model refactorings, some of them lead to simpler models, while others generalise or specialise concepts within a domain. Some multi-level refactorings are side-effect free, but others have semantic effects as they impact the features of existing instances of the refactored elements. To illustrate our catalogue, we use an example in the context of domain-specific process modelling. To demonstrate the feasibility of our approach, we provide an implementation of the refactorings in METADEPTH [de Lara and Guerra 2010], a textual multi-level modelling tool developed by our group.

The application of our refactorings is guided by the modeller, who decides where to apply them. This way, after the modeller selects a refactoring and the element to refactor, our implementation checks the refactoring pre-conditions, applies the refactoring, and repairs any inconsistent instance when the refactoring has side effects. Performing these tasks manually would be costly and error-prone. Heuristic mechanisms to automatically detect refactoring opportunities are planned for future work.

Multi-level models may become complex, so having a precise understanding of the consequences of their refactoring on existing and future instances is crucial. For this purpose, the definition of each refactoring states the side effects of its application, if there is any. To assess the predicted (or absence of) side effects, we have performed an experiment based on the generation of test models using model mutation, their automated refactoring, and the exhaustive bounded instantiation of the original and refactored models to detect side effects. Altogether, we have automatically performed more than 210.000 refactoring applications, and validated the semantic equivalence of the original and refactored models using more than 20.500.000 model instances. This approach we have taken for our evaluation can serve to other researchers as a basis to validate in-place transformations or refactorings for flat (i.e., non-multi-level) models.

In addition to side effects, refactorings may impact some quality attributes of a multi-level model, like simplicity, reusability or domain-specificity. To assess this impact, a second experiment measures the variation of internal metrics on multi-level models before and after the refactoring application. For this purpose, we used the 210.000 refactoring applications and the models of the first experiment. Our results help developers

in understanding the consequences of the chosen refactorings, and permit classifying refactorings according to their impact.

There is currently a vivid interest in multi-level modelling, as demonstrated by the MULTI workshop series¹, the recent Dagstuhl seminar on this topic [Almeida et al. 2017], and the plethora of emerging tools and frameworks like our own METADEPTH, DeepTelos [Jeusfeld and Neumayr 2016], DeepJava [Kühne and Schreiber 2007], DeepRuby [Neumayr et al. 2017], the DPF Workbench [Lamo et al. 2013], Dual Deep Modelling [Neumayr et al. 2018], Melanee [Atkinson et al. 2015], ML2 [de Carvalho and Almeida 2018; Fonseca 2017], MultiEcore [Macías et al. 2016], OMLM [Igamberdiev et al. 2018], SLICER [Selway et al. 2017], XMF and XModeler [Clark et al. 2015; Frank 2014]. We believe the catalogue we present in this paper can be a landmark to provide all these tools with advanced refactoring capabilities, needed to make multi-level modelling tools more mature.

The rest of the paper is organized as follows. First, Section 2 introduces multi-level modelling and a running example. Then, Section 3 provides a classification for multi-level refactorings, and Section 4 enumerates the refactorings in our catalogue, detailing four representative ones: *pull up clabject*, *extract clabject type*, *push down feature* and *replace reference by instantiation*. Next, Section 5 describes common themes and techniques used across refactorings. Section 6 presents tool support. Section 7 describes the experiments to assess the semantic side effects of each refactoring, and their impact on quality attributes. Finally, Section 8 analyses related works, and Section 9 concludes the paper. The appendix contains the detailed description of all refactorings except those presented in Section 4.

2. MULTI-LEVEL MODELLING

In this section, we introduce multi-level modelling with a focus on its architecture (Section 2.1), its main concepts through a running example (Section 2.2), its benefits (Section 2.3), and the challenges we aim to solve using refactorings, stressing the need for a catalogue (Section 2.4).

2.1. Architecture of multi-level frameworks

Multi-level modelling allows building modelling solutions where models are successively instantiated (or refined) in several consecutive meta-levels². It is a conservative extension of two-level modelling as, in the limit case, multi-level solutions which only involve two meta-levels are equivalent to standard two-level meta-modelling.

Multi-level modelling enhances flexibility by enabling an arbitrary number of meta-levels. It is appropriate in scenarios where the type-object pattern or some of its variants arise [de Lara et al. 2014], giving rise to smaller models than two-level solutions [Atkinson and Kühne 2008]. Our studies show that these scenarios are pervasive in domains like software architecture (where it is necessary to model component types and instances, port types and instances, and connector types and instances) and business process/enterprise modelling (e.g., to represent activity types and instances). They also occur in a significant number of OMG specifications (more than 35% of modelling specifications in year 2014) [de Lara et al. 2014].

Standard two-level modelling. Figure 1 shows the differences between two-level and multi-level modelling. Part (a) of the figure depicts a typical two-level architecture as advocated by the OMG’s Meta-Object Facility (MOF) [OMG 2016] or the Eclipse Modelling Framework (EMF) [Steinberg et al. 2008]. In this architecture, a meta-meta-model

¹<http://www.miso.es/multi>

²We use the term meta-level and level interchangeably.

defines a meta-modelling language (e.g., the *ecore* in case of EMF) that is used to build domain meta-models one level below. In their turn, meta-models can be instantiated to create models. The offered meta-modelling facilities (e.g., inheritance) are defined at the meta-meta-model and can only be used in meta-models. Implementation-wise, meta-models are usually compiled into classes of a programming language (e.g., Java in case of EMF), and models are created by instantiating those classes. Users can work with just two consecutive meta-levels at a time, and meta-models (typically built by language engineers) are frequently “hard” and compiled to enable model creation.

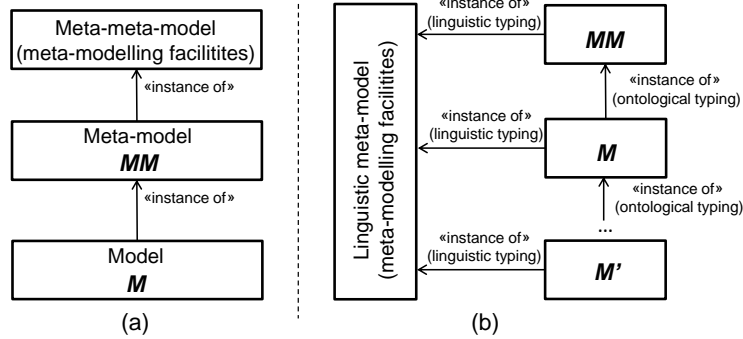


Fig. 1: (a) Standard meta-modelling architecture. (b) Multi-level, orthogonal classification architecture (adapted from [de Lara et al. 2014])

Orthogonal classification architecture: linguistic and ontological typings. Figure 1(b) shows a typical multi-level architecture that relies on the Orthogonal Classification Architecture (OCA) proposed by Atkinson and Kühne [Atkinson and Kühne 2002]. It distinguishes between linguistic typing (or physical dimension) and ontological typing (or logical dimension). Linguistic typing refers to the meta-modelling primitive used to create an element (model, clabject, field), wherein a linguistic meta-model makes all meta-modelling facilities available to all meta-levels [de Lara et al. 2015]. Ontological typing refers to the classification of an element within a domain. All meta-levels in a multi-level model are “soft”, and the ontological typing is reified and explicitly modelled in the linguistic meta-model, hence enabling an arbitrary number of meta-levels. This is the architecture we follow in this paper.

2.2. Multi-level modelling by example

As a running example, assume we need a system that allows describing process models for particular domains, like software engineering, learning management or business processes. Figure 2 shows a simplified multi-level solution, inspired by [de Lara et al. 2015]. The top-level model defines the concepts, properties and relations that process modelling languages in any domain have, like *TaskType* and different *GatewayTypes*. The model in the intermediate level is an instance of the top-level model, and defines kinds of tasks and gateways for a specific domain – in this case, the software engineering domain – like *Analysis* or *Coding* which are instances of *TaskType*. Finally, the bottom-level model is an instance of the middle model, and corresponds to a concrete software process model.

Clabjects. In a multi-level model, the elements in the top meta-level are pure types, the elements in the bottom level are pure instances, and the elements in the intermediate level have a type facet as they can be instantiated in the level below, as well as

an instance facet because they are instances of elements in the level above. For this reason, elements in the different levels are uniformly called *clabjects* (merging of the words *class* and *object*) [Atkinson and Kühne 2008].

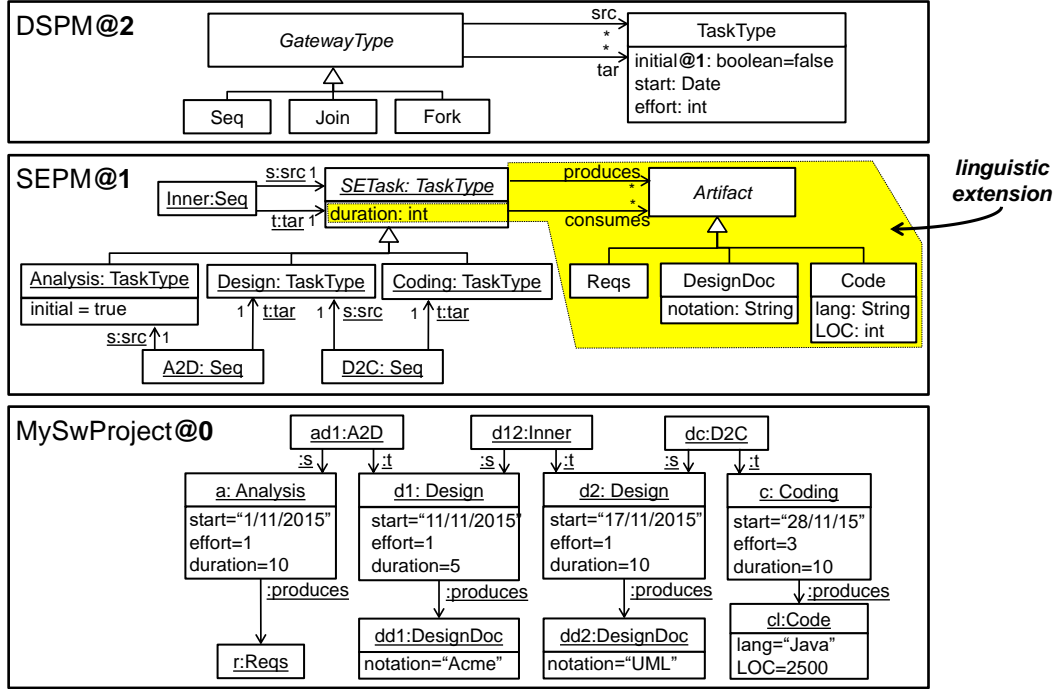


Fig. 2: Example multi-level model. Domain-specific meta-modelling language for process modelling (top); instantiation for software process modelling (middle); concrete software process model (bottom)

Potency. Models, clabjects, fields and references have an attached *potency* [Atkinson and Kühne 2002] to control their allowed instantiation depth. This is a number equal to or bigger than 0, which controls the number of successive levels in which an element can be instantiated. When an element is instantiated, the instance is automatically assigned the potency of the element minus 1. Elements with potency 0 cannot be instantiated in lower levels, because they are pure instances.

Figure 2 depicts potencies after the “@” symbol. The elements that do not declare a potency receive the potency of their immediate container (the owner class for fields and references, and the enclosing model for clabjects). For instance, the top-level model DSPM has potency 2, and hence, all its clabjects receive this potency by default so that their instantiation can only occur in the two subsequent levels. Alternatively, clabjects could have declared a potency lower than the model’s one. The potency of a model is sometimes called its level [Atkinson and Kühne 2008].

The potency of fields indicates how many levels below the fields can receive a value. This allows defining the properties of instances beyond the immediate lower level. In the top-level model of the figure, all fields receive potency 2 from their owner clabjects, except field *initial* which explicitly sets its potency to 1. In this way, fields *start* and *effort* in *TaskType* have potency 2, and therefore, any task at level 0 has these two properties.

Instead, field initial with potency 1 can only receive a value at the next level. This latter field is meant to signal the TaskType instances that are initial. For example, Analysis tasks are initial, but Design tasks are not.

Regarding references, they can declare a cardinality if their potency is bigger than 0, in which case, the cardinality constrains the instances at the next level below only. For example, GatewayType.src has potency 2 and cardinality *, while its instance Inner.s has potency 1 and cardinality 1..1. This means that, at the intermediate level, clobject Inner can have any number of references typed by GatewayType.src, but at the bottom level, d12 must have exactly one reference typed by Inner.s.

Linguistic extensions. The three-level solution in Figure 2 allows defining new process modelling languages for other domains (e.g., educational processes) as instances of the top-level model. Hence, one can see the top-level model as a domain-specific meta-modelling language for process modelling [de Lara et al. 2015]. However, some domains may require using concepts which either were not devised when designing the top-most level, or they are domain-specific and cannot be generalised (i.e., they do not belong to the top level). In such cases, it is still possible to extend models at any level including the bottom one with new clobjects, fields and references accounting for the new concepts. These are called *linguistic extensions* [de Lara and Guerra 2010].

In the example, the modeller added some linguistic extensions to the SEPM model, which we show inside a shaded polygon. Such extensions model new concepts specific to software engineering processes, like clobjects Reqs and Code, which are not instances of any top-level element. These two clobjects do not make sense at the upper level, as they do not apply to process models in other domains. The new field duration in SETask is another necessary linguistic extension as any software engineering task needs to have a duration.

Technically, linguistic extensions are possible by the *dual ontological-linguistic classification* for model elements introduced by the OCA, as depicted in Figure 1(b). Ontological classification represents instantiations within a domain, and linguistic classification refers to the meta-modelling primitive used to build a model element. For example, the ontological type of Analysis is TaskType, which is defined in the upper level, while its linguistic type is Clobject, which is defined by the meta-modelling framework. Hence, linguistic extensions are elements with linguistic type but no ontological type.

Inheritance and subtyping. The OCA also permits using inheritance at any meta-level, whenever the clobjects in the inheritance hierarchy have the same or covariant ontological type [de Lara et al. 2014]. For example, SETask and its children share the same ontological type TaskType. For fields with type facet, inheritance works as usual: the children clobjects receive the features defined by the parent clobject. For fields with instance facet, the children clobjects receive the value of the inherited fields and can override it, like in prototype-based languages [Noble et al. 1999]. Similar to the UML notation, clobjects with name in italics (e.g., GatewayType and SETask) are abstract and cannot be instantiated. Creating abstract clobjects, like SETask, as instances of non-abstract clobjects, is allowed.

2.3. Benefits of multi-level modelling

So far, we have seen that multi-level modelling provides flexibility on the number of meta-levels, supports language refinement by means of linguistic extensions, and allows using meta-modelling facilities like inheritance at any meta-level.

In comparison, using a two-level architecture to create a system like the one in Figure 2 would require workarounds that would introduce accidental complexity [Atkinson and Kühne 2008]. As an example, Figure 3 shows an excerpt of the running example using two-level and multi-level modelling. The two-level solution encodes DSPM as a

meta-model (MM in the architecture of Figure 1(a)), and both SEPM and MySwProject as an instance of it (M in Figure 1(a)). Elements instantiated at several levels in the multi-level solution need to be replicated using the type-object pattern [Martin et al. 1997] in the two-level solution. This is why DSPM in Figure 3(a) declares two classes to represent task types and their instances (TaskType and TaskInstance respectively). In addition, DSPM should be extended with further classes and features to allow task types to define inheritance hierarchies or declare task-specific properties and their cardinality (like SETask.duration in Figure 2). These linguistic facilities do not need to be modelled in a multi-level setting because they are globally available in the linguistic meta-model [de Lara et al. 2014]. As Figure 2 shows, the multi-level model has less elements than the two-level solution (3 objects vs 4 objects and 2 references).

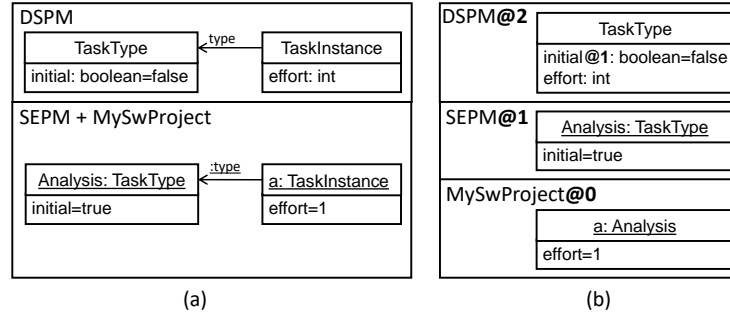


Fig. 3: Excerpt of running example using two levels (a) and three levels (b)

On the other hand, the standard two-level meta-modelling architecture of the OMG considers four meta-levels [Bézivin and Gerbé 2001], where the “real world” being modelled is placed at M0, right below M in Figure 1(a). Still, using this architecture, the three levels in Figure 2 should be shoehorned into two levels (MM and M). In contrast, multi-level modelling provides the ability to work with any number of levels. As an example, Figure 4 extends our running example with a fourth level in order to be able to describe processes in different software engineering subdomains, like web or mobile engineering. The figure shows the definition of the web engineering subdomain (WEPM) and an instance of it describing a specific web process (aWebProc). The example shows that the depth of a multi-level solution does not need to be decided a priori, but the potency of the top-level model can be left open (indicated by the potency “*” in DSPM). When instantiating elements with open potency, it is possible to assign them a concrete value (e.g., 2 in case of SEPM), or leave it open. Further examples of multi-level models with four or more levels can be checked out in [Rossini et al. 2015; Neumayr et al. 2018].

2.4. Relevance and need for a catalogue of multi-level refactorings

As Figures 2, 3 and 4 illustrate, multi-level modelling provides flexibility on the number of meta-levels a system spans, and linguistic extensions allow handling unforeseen or domain-specific concerns. However, at the same time, modellers have the burden to decide the level on which each model element should be defined. In Figure 2, the linguistic extensions (Artifact and its children, the duration field) would be unnecessary if the top-level model defined the concepts of *resource type* and *task duration*, which is sensible as most process modelling languages use some kind of resource, and tasks usually have duration. For example, the educational domain may define the task type Exam comprising the resource type Questionnaire. By creating in the top-level model a type ResourceType for Artifact, we would generalize a concept initially found in the

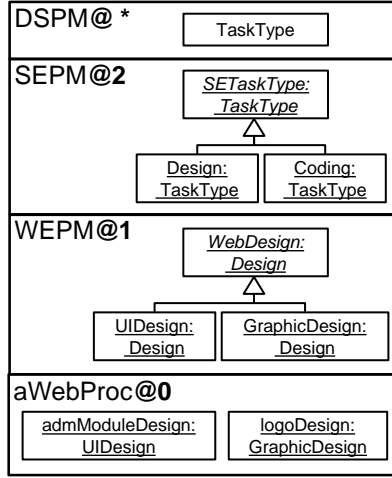


Fig. 4: Excerpt of multi-level model with four levels

software engineering domain to make it available to other domains, and enabling the creation of multiple types of resource within a domain. In contrast, field effort in the top level suggests specificity to domains where tasks are manual, and might be better placed at the middle level to make it specific to the software engineering domain. By relocating the field in this way, the generic process modelling language gets reduced, eliminating elements that only apply to a domain.

Figure 5 shows how the running example would be restructured by applying the above-mentioned changes (field duration is pulled up, field effort is pushed down, and a new clabject type for Artifact is extracted). The dashed coloured regions enclose the modified elements and have attached the name of the refactoring that would perform the changes.

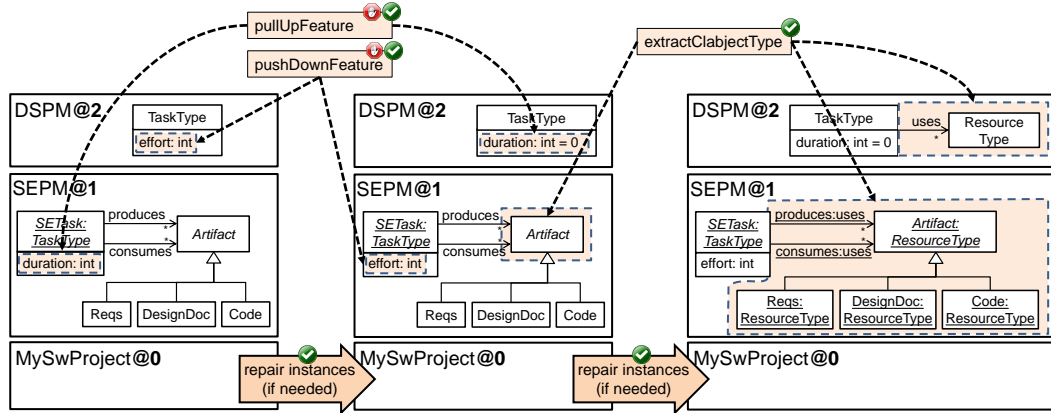




Fig. 5: Safe restructuring of running example using refactorings

Performing these restructurings by hand is tedious and error-prone, as some rearrangements require changing several parts of a model. For example, extracting the clabject type Resource Type for Artifact implies not only assigning a type to Artifact, but also

to all its subclabjects, common fields and incoming references (see right of Figure 5). If the modeller forgets performing any of these changes, the resulting multi-level model will be erroneous. Moreover, similar to model/meta-model co-evolution scenarios [Hebig et al. 2017], a change in a model may render invalid its instances at lower meta-levels, and so they need to be repaired. Doing so manually is costly and error-prone, especially if the number of instances to repair is large.

On the other hand, some multi-level restructurings may alter the set of instances a model accepts or rejects (i.e., the model semantics). In Figure 5, pulling field duration one level up makes any existing instance of TaskType to acquire a new slot for duration, changing its instance set. Hence, it is crucial that the modeller comprehends the implications of a restructuring before its application. We will discuss side effects more in detail in Section 3.2. In Figure 5, we mark refactorings that may potentially have side effects with symbol , and we use symbol  to indicate that, on this particular example, the refactorings have no secondary effects and there is no need to repair instances.

To facilitate exploring multi-level organization alternatives and help in the automation of rearrangements across levels, the remainder of this paper proposes multi-level refactorings which serve the following goals:

- Assess whether a multi-level restructuring application is safe (i.e., it does not yield incorrect models). In Figure 5, all restructurings can be safely applied.
- Report the consequences and side effects of a restructuring (e.g., clabjects that would gain extra fields). For example, pulling up a feature may have side effects in some cases, but not in the multi-level model of Figure 5.
- Perform the restructuring automatically, which may involve changing different parts of a multi-level model. In Figure 5, refactoring *extract clabject type* creates the clabject type ResourceType for Artifact and its subclabjects, and the reference type uses for references produces and consumes.
- Repair instances that become invalid as a result of a restructuring. In Figure 5, no repair action is needed.
- Improve some quality attribute of a multi-level model, like its reusability, domain-specificity or simplicity. Section 7.2 will analyse this aspect in more detail.

To rearrange the information in the multi-level models, our refactorings sometimes change instantiation by inheritance (e.g., an instance of a clabject becomes its subclabject after being promoted to an upper level), or vice versa (e.g., a subclabject is pushed one level down to be an instance). While inheritance is the main classification relation in standard two-level modelling, instantiation is the relation that permits layering a multi-level model. Our refactorings can also help in the migration of two-level designs into multi-level ones by converting type-object pattern instances into multi-level models (see Figure 3) or by replacing inheritance by instantiation.

3. MULTI-LEVEL MODEL REFACTORINGS: CLASSIFICATION AND SEMANTIC SIDE EFFECTS

Next, we present a classification of multi-level refactorings according to their application strategy, impact on the number of levels, granularity and availability of variants (Section 3.1). Then, we discuss the implications of having refactorings with semantic side effects (Section 3.2).

3.1. Classification of multi-level refactorings

We classify multi-level refactorings along several dimensions. Figure 6 shows the multi-level refactorings in our catalogue, their relations and their classification. To keep familiarity with standard refactorings [Fowler 1999], we have adopted some of their names, like *pull up feature*. However, multi-level refactorings impact several meta-

levels, while standard ones only impact one. For example, the standard *pull up feature* moves a feature to a superclass through the inheritance relation, while the multi-level counterpart moves a feature from a clobject to its type through the instantiation relation and may have side effects.

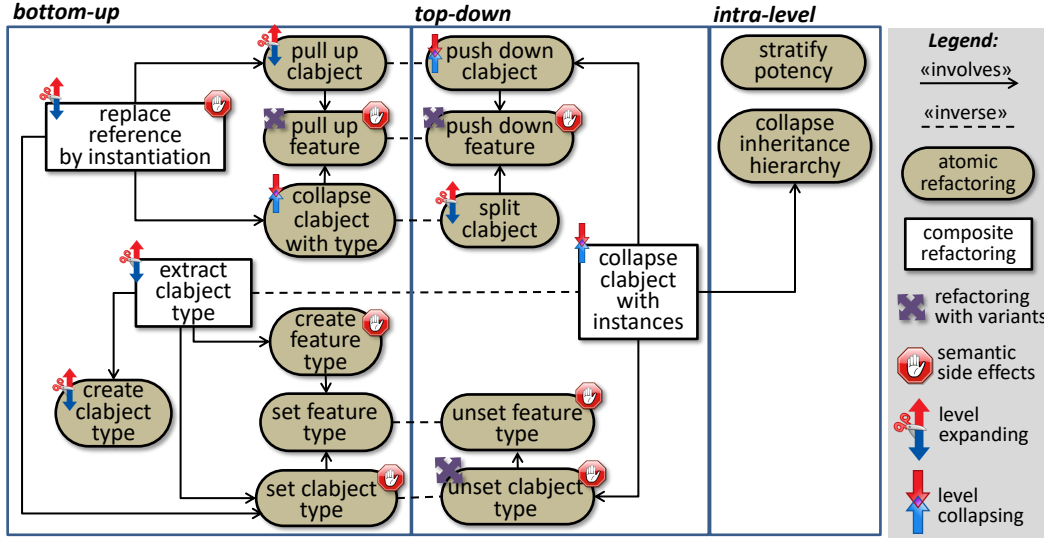


Fig. 6: Classification of multi-level refactorings

Multi-level refactorings can be generally classified as *bottom-up* if they create or move information to an upper level, *top-down* if they create or move information to a lower level, or *intra-level* if they rearrange information within a level. Standard model refactorings [Mens and Tourwé 2004] are a particular case of intra-level refactoring, though in a multi-level setting, they may need to be adapted to tackle discrepancies between the potencies or the ontological typing of the refactored elements. This classification also reflects two different modelling styles [Atkinson et al. 2011]: creating first types and then instances (top-down), or creating first instances and then types (bottom-up). The former is dominant in standard two-level modelling, while the latter permits an example-based approach to type construction [López-Fernández et al. 2015]. Both modelling styles are conveniently supported by refactorings in each category.

We also classify refactorings depending on whether their application may create new levels (*level expanding*), collapse two adjacent levels (*level collapsing*), or do not introduce new levels (*level preserving*). For example, the *pull up clobject* refactoring moves a clobject to a higher level, and when applied to a clobject defined in the top level, it creates a new level above to be able to pull up the clobject. Conversely, refactoring *collapse clobject with instances* merges a clobject with all its instances, which may imply collapsing two levels (in this case, by removing the upper level) if one of them becomes empty.

Some refactorings have an inverse that performs the reverse operation, like *pull up clobject* and *push down clobject* which are inverse of each other. The execution of a refactoring may require applying others. For example, pulling up a clobject also pulls up all its features. Moreover, we distinguish *atomic* and *composite* refactorings. The latter involve the coordinated execution of two or more atomic or composite refactorings. For

example, refactoring *collapse clabject with instances* consists of three atomic refactorings: *push down clabject* (which also makes its instances inherit from the pushed down clabject), *unset clabject type* of the instances, and then *collapse inheritance hierarchy*.

Finally, some refactorings have variants. For example, the default behaviour of refactoring *push down feature* is moving a feature to all instances of a clabject; alternatively, one can move the features to only a subset of the instances.

3.2. Side effects of refactorings

Code refactorings are meant to preserve behaviour [Fowler 1999]. As we deal with multi-level models, we do not have execution traces, but instead, we investigate the set of instances of the involved models. We do so because, from the point of view of the modeller, it is important to understand whether a refactoring alters the instances of the model being modified. Hence, we investigate *semantic side effects* of multi-level refactorings, that is, whether their application modifies the set of instances at any level (called *instance-data preservation* in [Noy and Klein 2004]). To analyse the effects of a refactoring at a certain level l , we check if the set of instances at lower meta-levels is preserved before and after the refactoring. Multi-level refactorings with side effects may need to repair existing instances, e.g., to add or remove fields. This is not necessary in standard model refactorings because they focus on one meta-level only [Weber et al. 2011], although the refactoring of interrelated models may require change propagation [Sunyé et al. 2001].

As an example, consider the *pull up feature* refactoring in Figure 7. The multi-level model (a-b-c) is based on the example in Figure 2: model (a) is an excerpt of DSPM, model (b) contains instantiations for the software engineering and educational domains, and model (c) has potency 0. The refactoring receives as parameters the elements where the refactoring is to be applied (SETask.duration in this case). Applying the refactoring to field SETask.duration yields the multi-level model (d-e-f). The refactoring pulls the field up to clabject TaskType in model (d), and this makes the EduTask instance in model (f) to receive a duration slot with default value 0 which did not have before. This means that the refactoring does not preserve the set of instances of model (b) at level 0.

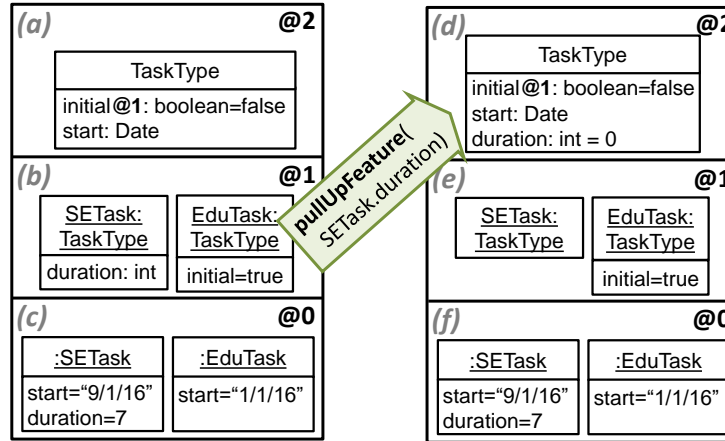


Fig. 7: Example of refactoring *pull up feature*: field duration is pulled up from SETask to TaskType

Figure 8 shows another example where model (c) does not conform to models (a-b), but does conform to the refactored models (d-e). This is so as clabject et has a field duration

(highlighted in the figure) which does not receive from EduTask in model (b), but receives it from clabject TaskType in model (d) (as duration has potency 2 in TaskType). Hence, the set of instances of (a-b) is not preserved, thereby we say the refactoring introduced a side effect. Section A.6 provides more details on the effects of this refactoring.

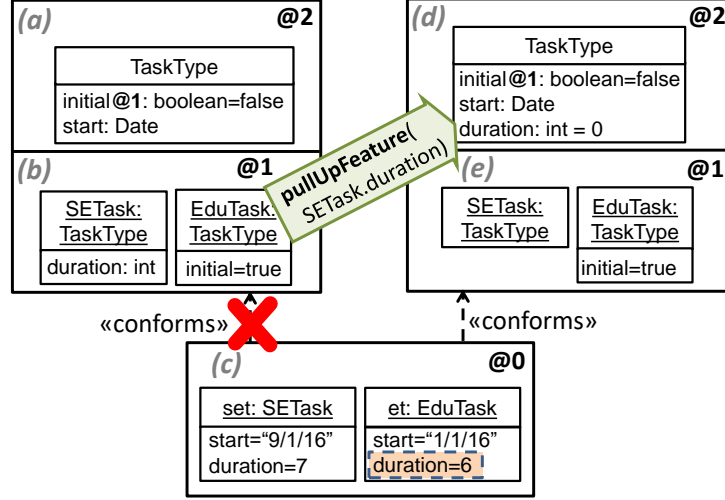


Fig. 8: Model showing that *pull up feature* does not preserve semantics

Side effects do not imply that a refactoring is incorrect, but they just signal consequences of its application. Still, refactorings should not create inconsistent models. For example, *pull up feature* requires assigning a default value to the pulled-up feature, or set it to optional, to keep the conformance of existing models. Otherwise, in Figure 7, any existing instance of EduTask would become inconsistent by lacking a slot for duration. Alternatively, the refactoring could create a slot with a default value in each existing instance of EduTask.

Hence, our refactorings tackle the meta-model/model co-evolution problem as follows: refactorings without side effects do not need to adapt the lower meta-levels where the refactoring is applied, while refactorings with side effects automatically co-evolve the lower levels.

4. CATALOGUE OF MULTI-LEVEL MODEL REFACTORINGS

Table I summarizes the 17 refactorings in our catalogue (see also Figure 6). The last column refers to the paper section where each refactoring is detailed.

We assume an unambiguous way to refer to the clabjects and features to which a refactoring is applied. In textual environments, this can be achieved by assigning a unique name to each clabject and feature, like in Figure 2. In graphical environments, one can click on the relevant elements. Refactorings take parameters, typically sets of these identifiers and new names for moved or created elements.

In the interest of brevity, this section does not detail all refactorings in the catalogue, but instead, we just describe some representative ones. For each refactoring, we provide: its type according to the classification of Section 3, its utility (scenarios where the refactoring is useful), an example based on Figure 2, variants (if any), consequences (considerations to take into account when the refactoring is applied), pre-conditions to apply the refactoring, description of conditions for having side effects (when possible),

Table I: Catalogue of multi-level refactorings, categorized according to their application direction (BU=bottom-up, TD=top-down, IL=intra-level), impact on the number of levels (LE=level-expanding, LC=level-collapsing, LP=level-preserving), granularity (A=atomic, C=composite), and semantic side effects (y=yes, n=no).

refactoring	description	direction	levels	granularity	side effects	section
1 <i>collapse clabject with instances</i>	Moves all features of a clabject to its instances, and removes the clabject.	TD	LC	C	n	A.1
2 <i>collapse clabject with type</i>	Moves all features of a clabject to its ontological type, and removes the clabject.	BU	LC	A	n	A.2
3 <i>collapse inheritance hierarchy</i>	Moves all features of a clabject to its direct children in the inheritance hierarchy, and removes the clabject.	IL	LP	A	n	A.3
4 <i>create clabject type</i>	Creates an ontological type for a clabject. The type is created with no features, so the features of the clabject become linguistic extensions.	BU	LE	A	n	A.4
5 <i>create feature type</i>	Creates an ontological type for a feature lacking one.	BU	LP	A	y	A.5
6 <i>extract clabject type</i>	Creates an ontological type for a set of clabjects. The created type includes the common features of the set of clabjects.	BU	LE	C	n	4.2
7 <i>pull up clabject</i>	Moves a clabject, its features and its incoming references to an upper level.	BU	LE	A	n	4.1
8 <i>pull up feature</i>	Moves a feature from an instance to its ontological type. By default, a feature can be pulled up only if all instances of the ontological type declare the feature. Alternatively, it could be present in a subset of the instances.	BU	LP	A	y	A.6
9 <i>push down clabject</i>	Moves a clabject and its features to a lower level.	TD	LC	A	n	A.7
10 <i>push down feature</i>	Moves a feature from a clabject to its instances (all instances in the main variant, a selection of instances in the alternative one).	TD	LP	A	y	4.3
11 <i>replace reference by instantiation</i>	Replaces a reference modelling a type-object relation [de Lara et al. 2014] within one meta-level, by instantiation. This is done by pulling up the element playing the role of type.	BU	LE	C	y	4.4
12 <i>set clabject type</i>	Assigns an existing ontological type to a clabject lacking one, and to all possible matching features of the clabject with respect to features of the type. The unmatched features of the clabject become linguistic extensions.	BU	LP	A	y	A.8
13 <i>set feature type</i>	Assigns an existing ontological type to a feature lacking one.	BU	LP	A	n	A.9
14 <i>split clabject</i>	Creates an instance of a clabject in a lower level, and moves all features with potency higher than 1 to the instance.	TD	LE	A	n	A.10
15 <i>stratify potency</i>	Reorganizes the features of a clabject according to their potency along an inheritance hierarchy of abstract clabjects with decreasing potency [de Lara et al. 2014].	IL	LP	A	n	A.11
16 <i>unset clabject type</i>	Removes the ontological type of a clabject, and consequently, the ontological type of all features within the clabject. The clabject becomes a linguistic extension. In the main variant, the clabject only retains the features it defines. In the alternative variant, it retains the features it defines, and also those declared by its type.	TD	LP	A	y	A.12
17 <i>unset feature type</i>	Removes the ontological type of a feature, which becomes a linguistic extension.	TD	LP	A	y	A.13

and the refactoring working scheme (description of the operations that the refactoring performs). As the reader will notice, the pre-condition of all refactorings includes some multi-level-specific aspect, such as requiring an element to have or not an ontological type, to have or not instances in lower meta-levels, or to have a specific potency value. The appendix includes the rest of refactorings not presented in this section.

4.1. Pull up clabject

This refactoring promotes a clabject, its features and its incoming references to the upper meta-level.

Type. Bottom-up; might be level expanding; atomic; no semantic side effects.

Utility. Some clabject needs to be made more general in order to become available for other models or domains. Once pulled up, any model management operation defined over the clabject becomes applicable to all its instances.

Example. In Figure 9, model (a) contains the definition of a common concept in any process modelling language, which is TaskType. Model (b) defines specific kinds of tasks for software process modelling, all of which inherit from the base class SETask. Since many domains will likely need tasks with duration, the refactoring moves class SETask to the upper level, renamed as DurableTaskType to make it domain-agnostic. Since this refactoring has no semantic side effects in models with potency 0, model (c) remains unchanged (see model (f)).

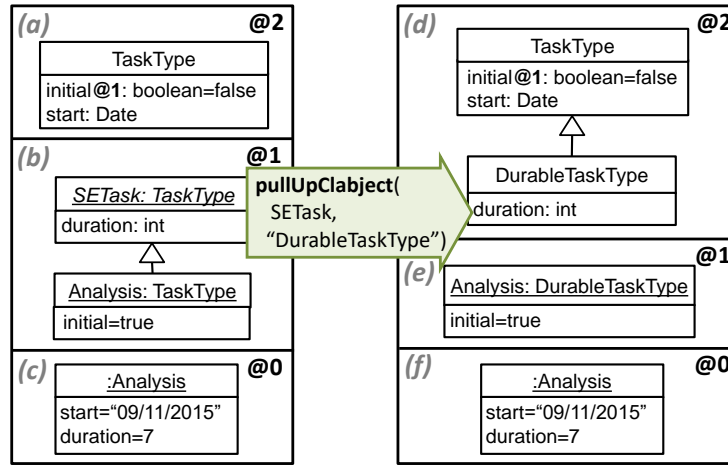


Fig. 9: Example of refactoring *pull up clabject*: clabject SETask is promoted one level up, renamed as DurableTaskType

Consequences. Promoting a clabject one level up makes it available for its instantiation in the level where it originally belonged, in addition to the levels where it could be instantiated before the refactoring. Thus, in Figure 9, DurableTaskType can be instantiated in model (e) and indirectly in model (f). Moreover, the pulled clabject maintains the values assigned to its slots. For example, if SETask provided a value for field initial in model (b), DurableTaskType.initial in model (d) would be assigned this value as default.

This refactoring is different from just creating a subclass of TaskType, as in that case, SETask would remain at level 1, and the type of Analysis would not change to DurableTaskType.

Pre-conditions. This refactoring cannot be applied to clabjects that refer to, or are referred by, clabjects without ontological type, as these references could not be pulled up. Similarly, the refactoring cannot be applied to clabjects that receive or own references that are instantiated at some level below, as pulling up such references would make the instances at the lower level incorrect. Clabjects with instances cannot be pulled up using this refactoring, but refactorings *create clabject type* or *extract clabject type* should be used instead.

Working scheme. Refactoring `pullUpClabject(C, name)`, where C is the clabject to pull up and name its new name, works as follows:

- (1) If C resides in a model M with no ontological type, then create a new model MM, and make MM the ontological type of M. This causes a level expansion.
- (2) Create a new non-abstract clabject C' in the ontological type of M.
- (3) Rename C' to name, and give it the potency of C plus 1.
- (4) Copy all fields from C to C', increasing their potency by 1, and assigning as default value the one they have in C.
- (5) Pull up each reference going from C to some clabject D. These references are pulled up to C' and set to point to the ontological type of D.
- (6) Pull up each reference going from some clabject D to C. These references are pulled up to the ontological type of D and set to point to C'.
- (7) If C has an ontological type, make C' inherit from it.
- (8) Make each subclass of C an instance of C' and a subclass of C's superclasses.
- (9) Remove C, its properties and its incident references from M.

In this algorithm, if C contained slots (i.e., fields with potency 0), they would be handled by step 4 (i.e., they would be copied to the new class C' with potency 1 and the slot content as default value). Although steps 4 and 5 can be performed using refactoring *pull up feature*, they are explicitly stated in the algorithm for clarity.

In model (d) of Figure 9, DurableTaskType is non-abstract (step 2), has potency 2 (step 3), defines a field duration with potency 2 (step 4), and inherits from TaskType (step 7). Subclasses of SETask (Analysis) become instances of DurableTaskType (step 8).

4.2. Extract clabject type

This refactoring creates a new ontological type for one or more clabjects lacking one. The created type contains the common features of all involved clabjects.

Type. Bottom-up; might be level expanding; composite; no semantic side effects.

Utility. Several untyped clabjects (linguistic extensions) are relevant at a higher level because they are useful in other domains, and they share a number of commonalities. Hence, the linguistic extensions are found to be instances of a common primitive, which is reified at the upper meta-level. This refactoring is also useful for bottom-up modelling [López-Fernández et al. 2015], where instances are created first in an exploratory way, and types are derived afterwards from the features of the instances.

Example. In Figure 10, model (b) defines the linguistic extension Artifact to model the resources that software engineering tasks can produce and consume, while the linguistic extension Environment refers to the development environment for coding. On reflection, the designer realises that both extensions correspond to different kinds of resources, and many domains will likely need their own domain-specific resources as well. By applying *extract clabject type* over Artifact and Environment, a new clabject ResourceType is created in the top level and becomes available to other domains. Moreover, references codedIn, produces and consumes become generalized by TaskType.uses (see model (d)). After applying this refactoring, model (c) remains unchanged as the refactoring has no side effects in the bottom level.

Consequences. The level above the selected linguistic extensions is expanded with a new primitive, which becomes their ontological type (i.e., they cease to be linguistic extensions). Common features to all selected extensions are pulled up to the new primitive. The primitive can be instantiated to create clabjects with structural similarity to the original linguistic extensions.

This refactoring is similar to Fowler's extract superclass [Fowler 1999]. However, while Fowler's uses the inheritance relation to create a superclass with the common fields of a set of classes, ours uses the instantiation relation to create a new type at a higher level.

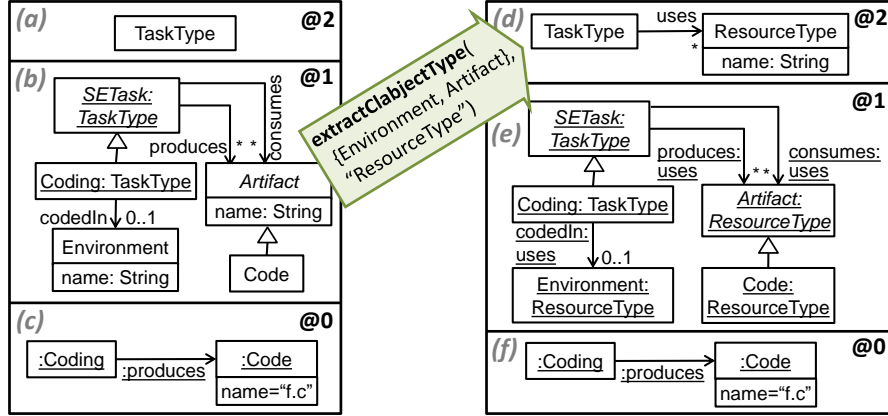


Fig. 10: Example of refactoring *extract clabject type*: a new clabject `ResourceType` is created at level 2, as a type for clabjects `Environment` and `Artifact` at level 1

Pre-conditions. This refactoring can only be applied to clabjects without an ontological type and without parent clabjects. Moreover, all refactored clabjects must have the same potency and reside in the same level.

Working scheme. Refactoring *extractClabjectType*($C=\{C_i\}_{i \in I}$, name), where C_i are the clabjects to generalize and name is the name of their new ontological type, works as follows:

- (1) If all C_i reside in a model M with no ontological type, then create a model MM , and make MM the ontological type of M . This causes a level expansion.
- (2) Create a new non-abstract clabject C' in the ontological type of M .
- (3) Rename C' to name, and give it the potency of the C_i clabjects plus 1.
- (4) Set the ontological type of every C_i and its children clabjects to C' .
- (5) Let $P=\{p_j\}_{j \in J}$ be the maximal set of common fields (with primitive type) for all clabjects in C .
- (6) For every $p_j \in P$:
 - (a) create a clone p'_j in C' , with its potency increased by 1.
 - (b) set the type of p_j in every C_i to p'_j .
- (7) For each reference r_j from a clabject $C_i \in C$ to some clabject D with ontological type:
 - (a) if no reference r exists from C' to the ontological type of D , with potency equals to the potency of r_j plus 1, then create one with cardinality $*$.
 - (b) set the type of r_j to r .
- (8) For each reference r_j from some clabject D with ontological type to a clabject $C_i \in C$:
 - (a) if no reference r exists from the ontological type of D to C' , and potency equals to the potency of r_j plus 1, then create one with cardinality $*$.
 - (b) set the type of r_j to r .

While this is a composite refactoring, the previous algorithm is presented in full detail for clarity, with no mention to the refactorings making up the composite. Steps 1 to 4 correspond to applications of the atomic refactoring *create clabject type*, which would be applied on one clabject $C_i \in C$. Then, steps 6 to 8 apply the atomic refactoring *create feature type* to each feature in P . Finally, we apply *set clabject type* to the other clabjects in C , to set their type to the newly created clabject, and also type their features.

In Figure 10, ResourceType in model (d) becomes the ontological type of Environment, Artifact and its child clabject Code (step 4). Step 6 is executed for field name, as this field is defined by all the clabjects to be generalised. Step 8 is executed for references codedIn, produces and consumes, for which the new reference type uses is created in the top level. The name of the created reference type can be asked to the user (see Section 6). The algorithm generalises the cardinality of the created reference type to *. While it can be adjusted to be the exact number of instances (3 in the example), generalising provides more flexibility.

4.3. Push down feature

This refactoring moves a feature (field or reference) from a clabject to all its instances. Alternatively, the variant moves the feature to a subset of instances.

Type. Top-down; level preserving; atomic; the alternative variant, which moves a feature to a subset of instances, has semantic side effects.

Utility. A feature deemed general across domains, is found to be specific to some particular domains or set of clabjects. Optional features which frequently do not get any value on instances may be indicative of this situation.

Example. In model (a) of Figure 11, TaskType defines the field effort with potency 2, assuming that all tasks two levels below will have a measurable effort. Since this may not be the case in all domains (e.g., in domains where tasks are automatic) or for all instances, this refactoring moves the field to the subset of instances of TaskType for which the field makes sense (in this example, to SETask). This yields the multi-level model (d-e-f). The application of this refactoring has semantic side effects, as PerformBackup and its instances in the bottom level defined the field before the refactoring, but not afterwards.

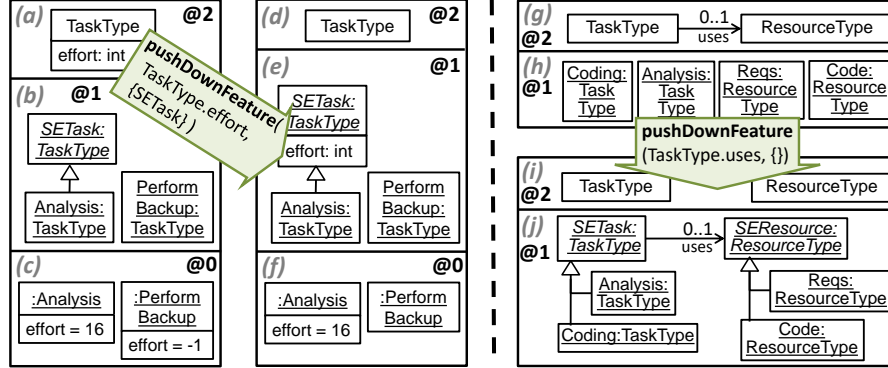


Fig. 11: Examples of push down feature: (a-f) primitive field, (g-j) reference

The right of Figure 11 shows another example, where the reference uses defined in model (g) is pushed down to all instances of TaskType. The refactoring introduces two abstract clabjects SETask and SEResource from which all instances of TaskType and ResourceType inherit, and which become the source and target of the pushed down reference. This variant of the refactoring, where a feature is moved to all instances of the clabject that defines the feature, has no semantic side effects.

Variants. In the main variant, the feature is pushed down to all instances of the clabject. In the alternative variant, a subset of the instances is selected.

Consequences. The clabject that initially defined the feature becomes more general and applicable to further domains. The alternative variant of this refactoring has semantic side effects, as any instance clabject not selected for pushing down the feature will lack the feature after the refactoring. As a consequence, some repair actions may be necessary to remove the lacking feature from these clabjects and their instances (see PerformBackup and its instance in the left refactoring of Figure 11 for an example). The refactoring performs these repairing actions automatically. While Fowler’s refactoring *push down field* [Fowler 1999] moves a field to a selected number of subclasses, our refactoring moves it to a subset of instances in a lower level and may require repairing actions for the non-selected instances.

Pre-conditions. The pushed down feature should have potency bigger than 0, and the owner clabject should have instances. If the feature is a reference, in addition, the target clabject should have instances, but not the reference.

Side effects. The refactoring has side effects if the feature is not pushed down to all instances of the feature owner (alternative variant). This is so as the instances where the field is not pushed down will lack the feature.

Working scheme. Refactoring pushDownFeature($C, f, D = \{D_i\}_{i \in I}$), where f is a feature owned by C , and D_i are the selected instances of C to push down f , works as follows:

- (1) If D is empty, initialize D with the set of all direct instances of C .
- (2) If $|D| > 1$ and the D_i do not inherit from a common clabject, then create an abstract clabject A instance of C , and set each D_i to inherit from A .
- (3) If $|D| = 1$ then let P be the unique D_0 in D . Else let P be the common ancestor A of all D_i .
- (4) Create a clone f' of f , with its potency decreased by 1. Assign f' to P .
- (5) If f is a reference pointing to a clabject E , then:
 - (a) if the instances of E do not inherit from a common clabject, then create an abstract clabject F instance of E , and set each instance of E to inherit from F . Else let F be the common ancestor of the instances of E .
 - (b) f' is set to point to F , and f' ’s cardinality is copied to f' .
- (6) Remove f from C . Remove the instances of f from any instance D_j of C with $D_j \notin D$, and so on recursively for their instances in lower levels.

If the algorithm receives an empty set of instances, it initializes set D with all instances of C , i.e., it applies the main variant of the refactoring (step 1). In the left refactoring of Figure 11, the field effort is moved to SETask as set D only contains this clabject (steps 3 and 4). Moreover, a repair action removes the field from PerformBackup and its instances in the bottom level (step 6). In the refactoring to the right, reference uses is pushed down to all instances of TaskType (steps 1, 3, 4 and 5). This induces the creation of two abstract clabjects (steps 2 and 5a).

4.4. Replace reference by instantiation

This refactoring replaces a reference modelling an *instance-of* relation by real instantiation. This explicit modelling of instantiation with a reference is known as the type-object pattern [de Lara et al. 2014; Martin et al. 1997]. The refactoring rearranges an occurrence of this pattern (i.e., two classes with roles *type* and *object* in the same level, related by a reference that emulates the *instance-of* relation) into a multi-level solution that makes use of the built-in instantiation relation provided by the meta-modelling framework. As described in our previous work [de Lara et al. 2014], the type-object pattern is recurrent when there is the need to emulate multiple levels within one. Thus, this refactoring helps in migrating two-level solutions into multi-level ones.

Type. Bottom-up; might be level expanding; composite; it has semantic side effects.

Utility. A reference emulating the instantiation relation is replaced by real instantiation to obtain a simpler modelling solution (less objects and references), or to benefit from the facilities that instantiation provides (e.g., conformance checking of instances with respect to types, ability to define constraints on types to be evaluated on their instances, automatic provision of slots in instances for all fields defined in their types, etc.) [de Lara et al. 2014].

Example. Model (a) in Figure 12 contains clajects TaskType and Task related by a reference emulating the instantiation relation. While TaskType declares the field initial instantiable on task types, Task declares field start instantiable on specific tasks. Model (b) instantiates model (a) to define the task type Analysis and one “instance” of it, emulated by a link between them. The refactoring replaces both references in model (a-b) by instantiation. For this purpose, TaskType is promoted one meta-level up with increased potency, holding field initial with potency 1 so that it is available on TaskType instances, and field start with potency 2 so that it is available at the bottom level. Analysis is promoted to level 1, and becomes the type of claject a. Altogether, the resulting model (c-d-e) has less elements.

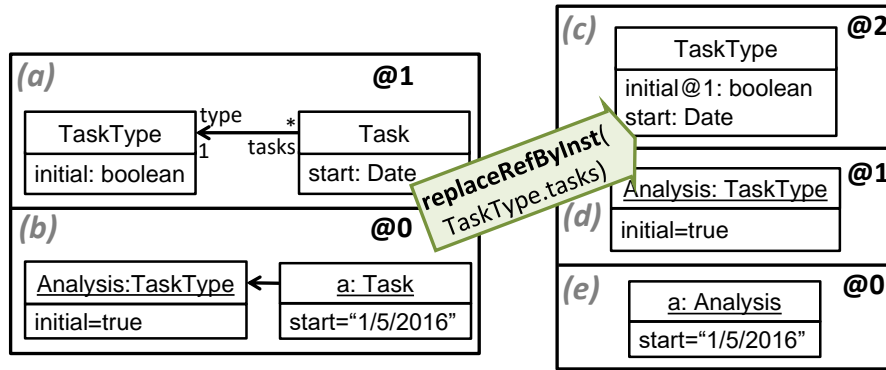


Fig. 12: (a-b) Type-object pattern. (c-d-e) Refactoring by *replace reference by instantiation*

Consequences. The resulting model uses the built-in instantiation relation in place of references. This leads to more meta-levels, but the top-level model is simpler as there is no need to define separate classes for types (e.g., TaskType) and objects (e.g., Task). For example, model (c) in Figure 12 has one claject, while model (a) has two clajects and a reference.

This refactoring is different from Fowler’s refactoring *in-line class* [Fowler 1999], which would merge TaskType and Task and does not consider multiple levels. Instead, *replace reference by instantiation* promotes TaskType and Analysis one level up, increases the potency of field start, and removes the reference type.

Pre-conditions. The claject playing the role of *type* should have the same potency as the claject playing the role of *object*, and cannot hold any reference apart from the one being replaced by instantiation, nor be the target of any reference. This is so as this claject and its (direct and indirect) instances will be pulled up to upper levels. Similarly, the claject with role *type* and its instances cannot have sub- or superclajects.

Side effects. This refactoring always has side effects, because it eliminates a reference and a claject. This way, instantiating the claject with *object* role (i.e., Task in Figure 12) leads to a model which is not accepted by the refactored solution.

Working scheme. Refactoring `replaceRefByInst(D.f)`, where D is a clabject playing the role of *object* in a type-object occurrence, and f is a reference owned by D to be replaced by instantiation, works as follows:

- (1) Let C be the target of D.f.
- (2) For each instance C_i of C:
 - (a) apply refactoring `setClabjectType(C_i , D)`.
 - (b) apply refactoring `pullUpClabject(C_i)`.
 - (c) for each clabject D_{ij} instance of D that is connected to C_i by an instance f' of f:
 - i. remove f' from D_{ij} .
 - ii. apply refactoring `setClabjectType(D_{ij} , C_i)`.
- (3) Remove f from D.
- (4) Apply refactoring `pullUpClabject(C)`, and decrease the potency of all fields in C by 1.
- (5) Apply refactoring `setClabjectType(D, C)`. Set D to abstract.
- (6) For each children clabject C_i of D:
 - (a) apply refactoring `setClabjectType(C_i , C)`.
- (7) Apply refactoring `collapseClabjectWithType(D)`.

For simplicity, the algorithm assumes D has potency 1. If it had higher potency, the refactoring should be applied to level 1, and then iterated to higher meta-levels. The algorithm in case the *instance-of* reference is defined in the *type* clabject instead of in the *object* clabject is similar.

Figure 13 illustrates the working scheme. Model (a) contains an occurrence of the type-object pattern, and model (b) has two instances of `TaskType` and `Task`. The strategy is producing a powertype as a first step (model (c-d)), unfolding it into a multi-level model (model (e-f-g)), and then making use of potency for fields (model (h-i-j)).

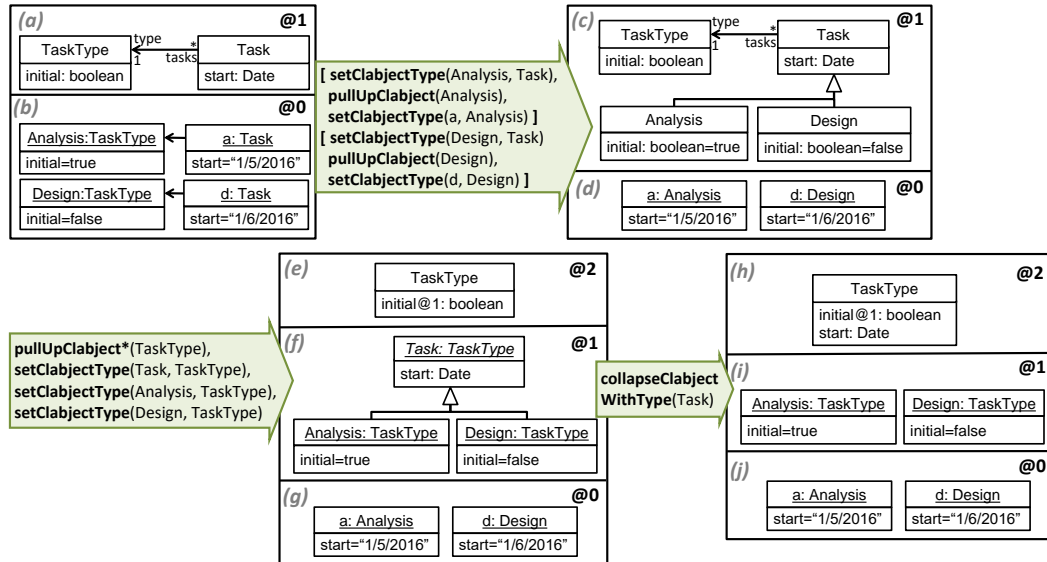


Fig. 13: Steps in the refactoring *replace reference by instantiation*

More in detail, we apply the refactoring `replaceRefByInst (Task.type)` to model (a), leading to model (h-i-j) as follows. First, the type of `Analysis` and `Design` is set to `Task` (step 2a), and both clabjects are pulled one level up, which makes them inherit from `Task` (step

2b, see Section 4.1). The type of a and d is set to Analysis and Design respectively (step 2c). The obtained model (c-d) reflects a powertype [Odell 1994] relation between the clabjects: TaskType is the powertype of Task because the original instances of TaskType become subclabjects of Task once pulled up. In step 4, the powertype is unfolded by promoting TaskType one level up (see model e). The potency of initial is set to 1 so that the instances of TaskType can give it a value (see model f). In step 5, Task is assigned type TaskType and made abstract. In step 6a, the type of Analysis and Design is also set to TaskType, and their field initial is retyped to TaskType.initial. These steps lead to model (e-f-g), where Task declares the field start that is common to Analysis and Design. While this could be the refactoring result, we choose to take advantage of potency, collapsing Task with its type (step 7). The resulting model (h-i-j) has less elements than (a-b) (5 clabjects w.r.t. 6, and 0 references w.r.t. 3). Moreover, the obtained solution is more flexible as one can add further fields to Analysis and Design without changing the top-level model, which is not possible in model (a-b).

5. COMMON THEMES AND TECHNIQUES USED IN REFACTORINGS

In this section, we analyse two recurring themes or techniques used across refactorings. On the one hand, in both bottom-up and top-down refactorings there is the need to replace instantiation by inheritance, or vice versa. On the other hand, some refactorings need to create “artificial” abstract clabjects, typically as the source or target of references being pulled up or pushed down. We call these clabjects “representatives”, and are an idiomatic form to represent a powertype relation.

5.1. Trading inheritance and instantiation

Some refactorings replace inheritance of clabjects by instantiation of clabjects (*push down clabject*), or instantiation of clabjects by inheritance of clabjects (*pull up clabject*), and some intermediate steps of *replace reference by instantiation*, see Figure 13).

For example, Figure 9 shows an application of the bottom-up refactoring *pull up clabject* to the clabject SETask, which gets promoted to a higher level. As a result, its type TaskType becomes its superclabject after the refactoring, and its subclabject Analysis becomes its instance.

Refactoring *push down clabject* performs the inverse operation, as Figure 14 illustrates. In this example, the modeller realizes that clabject SEDocumentation in model (a) is specific to the domain of software engineering and should be pushed one level down. After applying the refactoring, DocumentationType becomes the type of SEDocumentation, and is made concrete to allow its instantiation. Moreover, the instances of SEDocumentation in model (b) (like Diagram) become its children clabjects, which allows them to inherit field format.

There are also refactorings that move features between clabjects and their types, and hence, they interchange inheritance of features by instantiation of features. The consequence is that clabjects that originally inherited a feature end up receiving the feature through instantiation after applying the refactoring (e.g., in refactorings *pull up feature* and *collapse clabject with type*), or vice versa (e.g., in refactorings *push down feature*, *split clabject* and *collapse clabject with instances*).

As an example, the *push down feature* refactoring application in Figure 11 moves field TaskType.effort with potency 2 one level down. Hence, the instances of TaskType, which originally received the field through instantiation, receive it through inheritance after the refactoring.

5.2. Clabject representatives and the powertype pattern

Clabject representatives are abstract clabjects from which all instances of a given type inherit. Hence, they are an *idiom* to model the powertype pattern [de Lara et al.

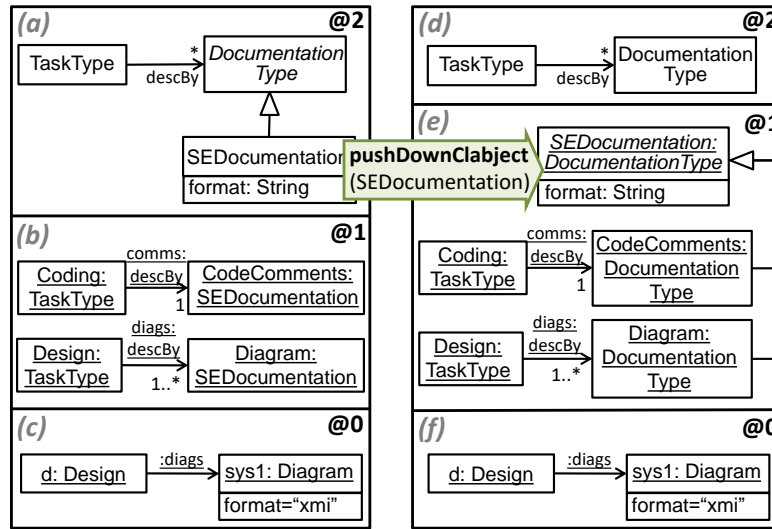


Fig. 14: Example of *push down clabject*: clabject SEDocumentation is pushed one level down

2014]. As an example, model (a) in Figure 15 shows an occurrence of the powertype pattern, while models (b-c) show the idiomatic representation of the pattern using representatives. In both cases, Analysis and Design are instances of TaskType and inherit from Task. However, in models (b-c), TaskType belongs to a higher meta-level, Task is its representative, and any instance of TaskType is required to inherit from Task. The advantage of this solution is that TaskType can define its own fields (like initial), and its instances (i.e., Analysis and Design) can assign them a value.

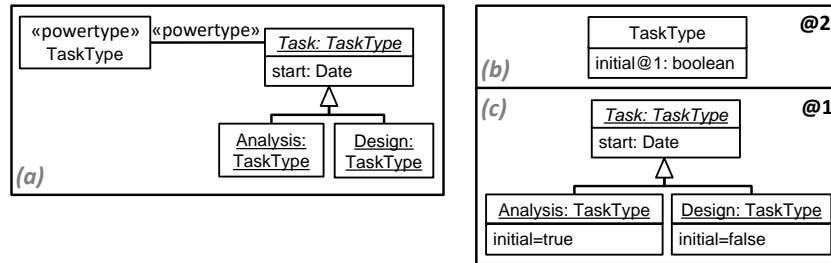


Fig. 15: (a) Powertype. (b-c) Its realization using clabject representatives

The following refactorings create clabject representatives: *push down feature*, *split clabject*, *push down clabject*, and intermediate steps of *replace reference by instantiation*. For example, when *push down feature* is applied to reference TaskType.uses in models (g-h) of Figure 11, two clabject representatives are created at level 1: SETask, which stores the pushed down reference, and SEResource, which becomes the reference target. In this way, the previously existing instances of TaskType get to inherit the reference from their representative. In the example of refactoring *replace reference by instantiation* in Figure 13, all instances of TaskType in model (f) inherit from Task, which is the clabject representative of TaskType. In Figure 14, refactoring *push down clabject* moves SEDocumentation one level below, becoming the representative of DocumentationType in

model (e). Although the refactoring makes all instances of SEDocumentation subclabjects of the representative, the designer could create further instances of DocumentationType that do not inherit from SEDocumentation. That is, clabject representatives work by convention, but are not enforced by the multi-level system. In order to enforce it, one could use refactoring *collapse clabject with type* to collapse the representative with its type. For example, in Figure 15, Task would be promoted and collapsed with TaskType, so that the instances of TaskType at level 0 would receive field start. This is actually done in the last step of refactoring *replace reference by instantiation* (see Figure 13).

6. TOOL SUPPORT

We have implemented the presented catalogue of refactorings in our multi-level modelling tool METADEPTH [de Lara and Guerra 2010]. This is a textual console-based tool which enables modelling at any number of levels, and supports dual ontological/linguistic typing and deep characterization through potency. It integrates the Epsilon languages for model management [Paige et al. 2009], which allows defining constraints, code generators, in-place and model-to-model transformations. METADEPTH supports derived attributes, the definition of methods for clabjects, and the analysis of constraints by model finding [Guerra and de Lara 2017].

The left of Listing 1 shows an excerpt of the multi-level model in Figure 2 using METADEPTH’s syntax. Lines 1–14 define the model DSPM with potency 2. All elements inside the model receive this potency, except field initial which declares potency 1. Lines 21–38 show an excerpt of the model SEPM one level below. It can be noted how instantiating a model has similar syntax as instantiating a clabject. Finally, lines 41–50 show an excerpt of the model MySwProject at level 0.

For this work, METADEPTH has been extended with commands to perform refactorings. The commands check the pre-condition associated to the selected refactoring and automatically rearrange the multi-level model, asking the user for any required information and reporting any possible semantic side effects.

As an example, typing `extractType ResourceType Artifact` in the console performs the refactoring described in Section 4.2, where the name of the new reference type (uses in the example) is prompted to the user. If instead of applying `extractType`, we try to apply `pullUpClabject` on `SETask`, the system informs that the refactoring cannot be applied because `SETask` has instances. If, after `extractType`, we perform the refactorings `pullUpFeature SETask.duration` and `pushDownFeature TaskType.effort`, we obtain the multi-level model in the right column of Listing 1 (also shown graphically in Figure 16).

To identify easily the elements affected by a refactoring, the tool annotates the refactored fields and clabjects. Annotations in METADEPTH have a similar syntax to Java annotations. For example, the annotation in line 77 of Listing 1 is attached to field `effort`, and indicates that the field was pushed down. Figure 16 shows these annotations using stereotypes.

Regarding side effects, neither `extractType` nor the main variant of `pushDownFeature` (which pushes a feature to all instances of the owner clabject) have side effects. In contrast, `pullUpFeature` may have secondary effects if the type where the feature is being pulled up (`TaskType` in our case) has other instances apart from `SETask` and its subclabjects. As this is not the case, the system issues no warnings, and hence the developer is ensured that the refactorings do not modify the model semantics.

7. EVALUATION

In this section, we evaluate two aspects of our refactoring catalogue. In the first place, we assess the side effects of our refactorings, stated in the *side effects* section of their definition. Hence, our goal is to answer the following research question (RQ1):

```

1  Model DSPM@2 {
2    Node TaskType {
3      initial@1 : boolean = false;
4      start : Date;
5      effort : int;
6    }
7    abstract Node GatewayType {
8      src : TaskType[*];
9      tar : TaskType[*];
10   }
11   Node Seq : GatewayType;
12   Node Join : GatewayType;
13   Node Fork : GatewayType;
14 }
15
16
17
18
19
20
21 // Instance of DSPM -----
22 DSPM SEPM {
23   abstract TaskType SETask {
24     duration : int;
25     produces : Artifact[*];
26     consumes : Artifact[*];
27   }
28   TaskType Analysis : SETask {
29     initial = true;
30   }
31   ...
32   abstract Node Artifact;
33   Node Code : Artifact {
34     lang : String;
35     LOC : int;
36   }
37   ...
38 }
39
40
41 // Instance of SEPM -----
42 SEPM MySwProject {
43   Analysis a {
44     start = "1/11/2015";
45     effort = 1;
46     duration = 10;
47     ...
48   }
49   ...
50 }
51
52 Model DSPM@2 {
53   Node TaskType {
54     initial@1 : boolean = false;
55     start : Date;
56     @extractType
57     uses : ResourceType[*];
58     @pullUpFeature
59     duration : int;
60   }
61   @extractType
62   Node ResourceType;
63   abstract Node GatewayType {
64     src : TaskType[*];
65     tar : TaskType[*];
66   }
67   Node Seq : GatewayType;
68   Node Join : GatewayType;
69   Node Fork : GatewayType;
70 }
71
72 // Instance of DSPM -----
73 DSPM SEPM {
74   abstract TaskType SETask {
75     produces : Artifact[*] { uses };
76     consumes : Artifact[*] { uses };
77     @pushDownFeature
78     effort : int;
79   }
80   TaskType Analysis : SETask {
81     initial = true;
82   }
83   ...
84   abstract ResourceType Artifact;
85   ResourceType Code : Artifact {
86     lang : String;
87     LOC : int;
88   }
89   ...
90 }
91
92 // Instance of SEPM -----
93 SEPM MySwProject {
94   Analysis a {
95     start = "1/11/2015";
96     effort = 1;
97     duration = 10;
98     ...
99   }
100   ...
101 }

```

Listing 1: Multi-level model (left) and refactored model (right)

RQ1: Do the refactorings in the catalogue yield the predicted side effects?

Side effects are an important aspect of our refactorings because they inform the modeller of the impact of applying a refactoring on existing instances. If a refactoring has side effects, it means it changes the set of instances of the refactored model, e.g., by adding or removing fields from the instances. Hence, it is essential to ensure that side effects are correctly identified for the catalogue to be useful.

Secondly, we assess the effect of refactorings on the quality of multi-level models. Therefore, our aim is to answer the following research question (RQ2):

RQ2: What is the impact of refactorings on quality attributes of multi-level models?

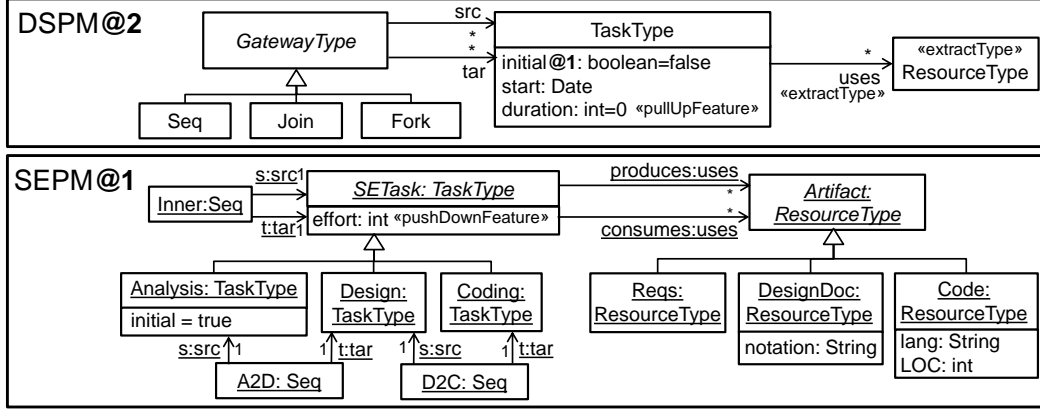


Fig. 16: Two upper levels of the refactored multi-level model

For this purpose, we propose metrics to determine three quality attributes of multi-level models (reusability, domain-specificity and simplicity) and observe the impact of refactorings on these metrics.

7.1. RQ1: Assessing refactoring side effects

To perform this evaluation, we use a technique based on model mutation and exhaustive model instantiation, inspired by [Daniel et al. 2007]. Roughly, the idea is generating a wide set of models by model mutation, to which we automatically apply the refactorings whenever their pre-conditions hold, ensuring that all corner cases are covered. Then, we check whether the refactoring applications modify the semantics of the multi-level models, in which case, we analyse whether the side effects are the ones we had predicted. Next, we describe the experiment setting (Section 7.1.1), its results (Section 7.1.2) and threats to validity (Section 7.1.3).

7.1.1. Experiment design and setting. Figure 17(a) shows the main steps in the evaluation. First, each refactoring being evaluated contributes several seed models. These are small multi-level models with situations of interest, which satisfy the refactoring pre-conditions (that is, they enable the refactoring application). Then, these models are mutated using sequences of mutation operations up to a given length. Table II describes the considered mutations. All mutations are additive, that is, they add information to the model, but do not change or destroy the original intent encoded in the seeds. The rationale is that, if the refactoring is applicable to a seed model, it will be likely (but not necessarily) applicable to its mutants. As a result of the mutations, we obtain a large population of test models. The refactoring under test is then applied on each such model m , leading to a refactored model m' . The goal of the experiment is to show that both models m and m' have the same semantics when the refactoring is meant to preserve the semantics, or to identify the situations where the refactoring does not preserve it. For this purpose, we systematically instantiate m and m' up to a given bound, giving rise to sets $SEM(m)$ and $SEM(m')$ of valid instances of m and m' , respectively. For m and m' to be semantically equivalent, m needs to accept $SEM(m')$, and m' should accept $SEM(m)$.

In order to answer RQ1, we measure, for each refactoring r , the number of pairs of original and refactored models $\langle m, m' \rangle$ in which some model in $SEM(m)$ is not accepted by m' , or some model in $SEM(m')$ is not accepted by m . There is a discrepancy if this

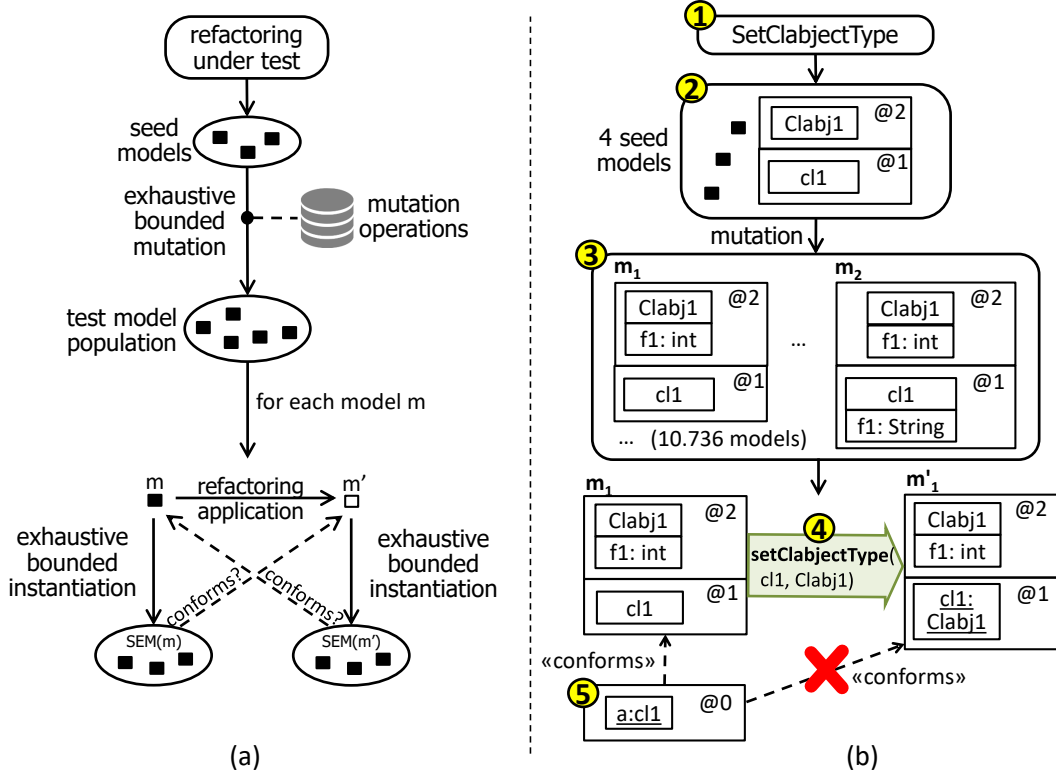


Fig. 17: (a) Steps to evaluate the side effects of a refactoring. (b) Evaluating refactoring *set clabject type*

Table II: Multi-level model mutations

Name	Description
CreateField	Creates a new field in a random clabject
CreateReference	Creates a new reference in a random clabject
CreateSubClabject	Creates a new clabject, child of a random clabject
CreateSuperClabject	Creates a new clabject, parent of a random clabject
InstantiateClabject	Creates an instance of a random clabject
InstantiateReference	Creates an instance of a random reference type

number is 0 but we predicted side effects for r ; or if this number is positive but we did not predict side effects for r .

Figure 17(b) illustrates this process for the evaluation of refactoring *set clabject type*. This refactoring (detailed in appendix A.8) assigns an existing type to an untyped clabject, as well as to as many compatible fields of the clabject as possible. We used four seed models in the experiment, one of which is shown in the figure with label 2. The refactoring is applicable to this model, as it can be used to type $cl1$ by $Clabj1$. We applied mutation operators randomly to the four seed models, obtaining 10,736 models. Label 3 in the figure shows two of them: m_1 and m_2 . In model m_1 , the mutation operator *create field* added the integer field $f1$ to $Clabj1$. In model m_2 , in addition, a string field $f1$ was added to $cl1$. The refactoring is still applicable to m_1 , but not to m_2 because typing $cl1$ by $Clabj1$ would cause a clash between $cl1.f1$ with type `String` and $Clabj1.f1$ with type `int`. Hence, we only apply the refactoring to m_1 and obtain model m'_1 , where $cl1$ gets typed

by Clabj1 (see label 4). To discover possible semantic side effects, we generate instances of m_1 and check if they conform to m'_1 , and we generate instances of m'_1 and check if they conform to m_1 . Label 5 in the figure shows an instance of m_1 which contains one clabject a. This model is not conformant to m'_1 because it lacks field f1, defined in Clabj1 with potency 2. This is a side effect correctly identified by the refactoring definition (see appendix A.8).

The experiment was performed using METADEPTH 0.3³, enlarged with a program specifically created to carry out the steps in Figure 17(a). To generate the set $SEM(-)$, we chose a bound of 3 instances per type and reference, and to generate the test model population, we applied all possible mutation sequences of lengths 1, 2 and 3 using the mutation operators of Table II. To discard errors in the refactoring application, a random 1% sample of the applied refactorings were manually revised independently by both authors (i.e., a total of 2.137 applications). The experiments were run on a PC with an intel Core i7 4770 and 16Gb RAM. The seed models used are available at <http://miso.es/ml-refacts>.

7.1.2. Experiment results. Table III summarizes the results of the experiment. The first three columns of the table contain the refactoring being evaluated, the number of seed models, and the number of test models generated from the seeds. We tried to apply the refactoring to each generated test model; hence, the next two columns show the number of times the refactoring could not be applied as the test model did not satisfy the refactoring pre-conditions (e.g., as model m_2 in Figure 17(b)), and the number of refactoring applications (i.e., number of test models minus number of failed refactoring applications). The remaining columns show the percentage of times the refactoring could not be applied, the number of instances used to check side effects, the number of refactored test models in which a side effect was found, and the percentage of refactored test models with side effects with respect to the total number of test models.

Table III: Results of the evaluation of refactoring side effects

Refactoring	#Seeds	#Models	#Failed applics.	#Applics.	% Failed applics.	#Instances	#Side effects	% Side effects
CollapseClabjectWithInst.	3	12.140	10.679	1.461	87,97%	160.644	0	0%
CollapseClabjectWithType	4	24.321	9.440	14.881	28,82%	374.550	0	0%
CollapseInheritHierarchy	3	72.542	0	72.542	0%	5.782.752	0	0%
CreateClabjectType	2	3.198	1.336	1.862	41,78%	121.452	0	0%
CreateFeatureType	3	5.227	0	5.227	0%	372.168	421	8,05%
ExtractClabjectType	4	25.635	0	25.635	0%	4.779.180	0	0%
PullUpClabject	4	11.751	10.426	1.325	88,73%	227.313	0	0%
PullUpFeature	3	7.812	5.194	2.618	66,49%	157.314	870	33,23%
PushDownClabject	3	23.715	9.922	13.793	41,84%	2.016.756	0	0%
PushDownFeature (main)	3	15.656	0	15.656	0%	2.166.834	0	0%
ReplaceRefByInstantiation	2	5.498	3.952	1.546	71,89%	85.851	1.546	100%
SetClabjectType	4	10.736	820	9.916	7,64%	684.096	3.187	32,14%
SetFieldType	3	5.505	0	5.505	0%	376.077	0	0%
SplitClabject	3	9.947	0	9.947	0%	685.722	0	0%
StratifyPotency	3	14.760	0	14.760	0%	1.332.468	0	0%
UnSetClabjType (main)	3	5.471	3.302	2.169	60,36%	92.163	84	3,87%
UnSetClabjType (variant)	3	5.471	3.340	2.131	61,05%	103.560	0	0%
UnSetFeatureType	4	12.711	0	12.711	0%	984.021	1.216	9,57%
Total:	57	272.096	58.411	213.685	21,47%	20.502.921	7.324	3,43%

Overall, refactorings were applied over 213.000 times, and semantic effects were investigated using more than 20 million model instances. In average, we generated 15.116 test models per refactoring, and 4.773 test models per seed. Each refactoring was

³Available at <http://metadepth.org>.

applied 11.871 times in average, with a median of 7.711. Around 96 instance models were used to validate the side effects of each refactoring application, and no unexpected side effect was found. Moreover, the high number of refactoring applications suggests that all corner cases were tested, providing high confidence in the validation of the predicted refactoring side effects.

In the table, column “% Side effects” gives an indication of the likelihood a refactoring application has side effects. The most side effect prone refactoring is *ReplaceRefByInstantiation* (see Section 4.4) with 100% of cases entailing side effects, as expected. This is so as the refactoring makes a heavy reorganization of the multi-level model, removing a reference and a clabject, and pulling several clabjects up.

Column “% Failed applics.” gives an intuition of the difficulty to meet the pre-conditions of each refactoring. Note that even if the original seed model satisfied the pre-conditions, the mutants generated may not meet them. The refactorings with the most restrictive pre-conditions are *PullUpClabject* (88,73%), *CollapseClabjectWithInstances* (87,97%) and *ReplaceRefByInstantiation* (71,89%).

No refactoring application had unexpected side effects, different from those predicted in the refactoring definitions. Hence, we can answer RQ1 affirmatively.

7.1.3. Threats to validity. While the presented experiment validates the assumptions on refactoring side effects by performing a large number of refactoring applications, it does not constitute a formal proof. For that purpose, formal approaches, for example based on constraint solving, could be followed. However, our own experience encoding some of the refactorings in Alloy [Jackson 2002] agrees with the assessment of other researches: formal proof of refactoring correctness is very hard [Estler and Wehrheim 2008]. In particular, the scalability of model finders to prove the general correctness of multi-level model refactorings (i.e., applied to any possible test model, not just on a particular model) is – at the time of writing this paper – not enough for this formal proof. However, the high number of generated test models enabled more than 213.000 refactoring applications, which provides a high confidence in the correctness of the results.

Another threat is that the chosen seed models might be biased towards cases leading to the predicted side effects, or to side-effect free refactoring applications. To alleviate this problem, we applied sequences of mutation operators (up to length 3) leading to the creation of around 4.700 test models per seed. Therefore, this effect can be neglected.

An additional issue is whether some refactoring produced side effects different from the ones predicted. To mitigate this possibility, each refactoring implemented a mechanism to discern whether the non-conformity of an instance was due to the expected side effect. We also inspected manually 1% sample of the refactoring applications to make sure they did correctly their work.

Another possibility is that some side effect may have not been caught because it occurs on model instances outside the search scope (3 instances of each clabject and reference). As our models do not have OCL constraints, we can discard this possibility because side effects typically appear on single clabject instances due to errors in its features, or in references connecting two clabjects. Hence, a bound of 3 is enough in this case.

The experiment considers models with three meta-levels, which is enough for checking side effects as these would appear already in three levels. This is so as, if we modify level 2 of a multi-level model, and the instances at level 1 before and after the modification are the same, then the instances at level 0 before and after need to be the same.

Finally, our experiment evaluates side effects, but not performance or scalability. Anyway, performance is not a concern because the user indicates where to apply the refactoring, the application execution time is negligible (at most linear on the model

size when the refactoring needs to repair instances), and the scalability is good. For example, a refactoring like *push down feature*, which needs to repair instances, has an average execution time around 10 milliseconds for instance models in the order of ten thousand elements.

7.2. RQ2: Impact of refactorings on quality attributes

Each refactoring definition includes a *utility* section explaining when to apply the refactoring. To further help modellers in understanding the implications of using a refactoring, we have conducted an experiment to assess how each refactoring affects three quality attributes of multi-level models. In the following, Section 7.2.1 introduces the considered quality attributes, Section 7.2.2 explains the experiment setting, Section 7.2.3 analyses the results, and Section 7.2.4 discusses threats to validity.

7.2.1. Quality attributes for multi-level models. Table IV summarizes the considered quality attributes for multi-level models, as well as the metrics used as indicators for them.

Table IV: Quality attributes and metrics for multi-level models

Attribute	Description	Metric
Reusability (RE)	Degree in which a model m can be reused to create models one level below	Size of model m (clabjects and fields)
Domain-specificity (DS)	Degree in which a model m contains domain-specific elements, not expressible with the primitives offered by the modelling language one level above	Ratio of linguistic extensions in m (clabjects and fields) w.r.t. size of model m (clabjects and fields)
Simplicity (SIM)	Overall simplicity of a multi-level model	Size of models in all levels (clabjects and user-visible fields)

The *reusability* (RE) of a model at a given level is the degree in which it can be reused to create instance models one level below. For example, in Figure 2, the DSPM model at level 2 is expected to be reused to create process models in different domains, like software engineering (model SEPM in the same figure), education or manufacturing. We use the model size (number of clabjects and fields) as a metric for this quality attribute. This metric has been used to quantify the reusability of design models [Bansiya and Davis 2002] and multi-level models [Gerbig 2017] with the rationale that the more classes in a design, the more reuse opportunities in different settings. While large meta-models (like that of UML [UML 2017]) tend to be highly reusable in many contexts, in our setting, decreasing reusability may be appropriate if the goal is eliminating unnecessary concepts which constitute an overgeneralization.

The *domain-specificity* (DS) of a model is calculated as the ratio of linguistic extensions in the model with respect to the model size. The more linguistic extensions, the more domain-specific concepts the model contains, which cannot be expressed with the generic primitives offered by the upper level. This metric takes values between 0 (no linguistic extensions) and 1 (all model elements are linguistic extensions). The metric can be seen as the inverse of the *recall* metric used to measure ontologies [Guarino 2004; Yu et al. 2009]. Under this perspective, recall would be an indicator of how good a classifier a meta-model is for the model at the level below, where 1 – a perfect classification – means no linguistic extensions, and 0 means no model element is typed.

The last quality attribute we consider is *simplicity* (SIM). As indicator of this attribute we compute the size of the whole multi-level model, measured as the number of clabjects and user-visible fields (i.e., fields that are either untyped or have a value). For example, in Figure 16, field TaskType.start is user-visible, but field Analysis.start is not user-visible because it does not assign to the field a value. In this case, the lower the metric (i.e., the lower the size), the simpler the model. Size has also been used in [Atkinson and

Kühne 2008; Atkinson and Kühne 2017] as an indicator of the complexity of multi-level models.

7.2.2. Experiment design and setting. To measure the impact of our refactorings on the identified quality attributes, we took the models that enabled the application of refactorings in the first experiment, computed the metrics before and after applying the refactoring, and assessed whether the metric value increased or decreased.

7.2.3. Experiment results. Table V gathers the results of the experiment. The first column contains the refactoring names and their abbreviation. Column *Intent* displays the goal of each refactoring in terms of the three quality attributes, using \uparrow to indicate that the attribute value is expected to increase, \downarrow if it is expected to decrease, and a blank cell if the attribute is not a concern of the refactoring. For example, *create clabject type* creates a clabject in an upper level, to act as the type for an untyped clabject in the lower level; hence, the refactoring aims at making the upper level more reusable by adding a new concept (RE= \uparrow), the lower level less domain-specific by acknowledging that the untyped clabject is an instance of a general concept existing at a higher level (DS= \downarrow), while simplicity is not a concern. Column *Real* shows the actual qualitative variation observed in the experiment. In this case, the symbol $-$ indicates no variation, while \sim represents very low or negligible variation. Finally, the remaining columns provide the quantitative variation obtained in the experiment for each attribute (interval of variation and average variation). For simplicity, these quantitative metrics aggregate the number of both clabjects and features.

Table V: Impact of refactorings on the quality of multi-level models. Columns show refactoring name; intended and real qualitative impact (\uparrow for increment, \downarrow for decrement, $-$ for no variation, \sim for negligible variation); and quantitative impact (variation interval $[min; max]$ and average variation).

Refactoring	Intent			Real			RE		DS		SIM	
	RE	DS	SIM	RE	DS	SIM	top-model size	linguistic ext.	overall size	linguistic ext.	overall size	overall size
CollapseClabjectWithInst. (CWI)	\downarrow	\uparrow		\downarrow	\uparrow	\downarrow	[-5; -2]	-2,27	[0,14; 0,86]	0,52	[0; 5]	1,81
CollapseClabjectWithType (CWT)	\uparrow	\downarrow		\uparrow	\downarrow	\uparrow	[1; 8]	1,95	[-0,71; 0,22]	-0,1	[-2; 2]	-0,99
CollapseInheritHierarchy (CIH)			\uparrow	\downarrow	$-$	\uparrow	[-1; 5]	-0,94	[0; 0]	0	[-1; 5]	-0,94
CreateClabjectType (CCT)	\uparrow	\downarrow		\uparrow	\downarrow	\downarrow	[1; 1]	1	[-0,33; -0,12]	-0,2	[1; 1]	1
CreateFeatureType (CFT)	\uparrow	\downarrow		\uparrow	\downarrow	\downarrow	[1; 1]	1	[-0,5; -0,08]	-0,16	[0; 1]	0,8
ExtractClabjectType (ECT)	\uparrow	\downarrow	\uparrow	\uparrow	\downarrow	\downarrow	[1; 8]	3,98	[-0,93; -0,2]	-0,56	[-3; 5]	0,76
PullUpClabject (PUC)	\uparrow	\downarrow		\uparrow	\downarrow	$-$	[2; 6]	2,75	[-0,9; 0,17]	-0,27	[0; 0]	0
PullUpFeature (PUF)	\uparrow	\downarrow		\uparrow	\downarrow	$-$	[1; 1]	1	[-0,5; -0,05]	-0,17	[0; 0]	0
PushDownClabject (PDC)	\downarrow	\uparrow		\downarrow	\uparrow	\sim	[-9; -1]	-3,73	[-0,03; 0,96]	0,64	[0; 1]	0,003
PushDownFeature (main) (PDF)	\downarrow	\uparrow		\downarrow	\uparrow	\downarrow	[-1; -1]	-1	[0,08; 0,5]	0,19	[0; 6]	2,21
ReplaceRefByInstantiation (RRBI)			\uparrow	\uparrow	\downarrow	\uparrow	[3; 6]	3,6	[-1; -1]	-1	[-5; -2]	-2,28
SetClabjectType (SCT)		\downarrow		$-$	\downarrow	$-$	[0; 0]	0	[-1; -0,14]	-0,45	[0; 0]	0
SetFieldType (SFT)		\downarrow		$-$	\downarrow	\uparrow	[0; 0]	0	[-0,33; -0,07]	-0,14	[-1; 0]	-0,15
SplitClabject (SC)	\downarrow	\uparrow		\downarrow	\uparrow	\downarrow	[-5; 0]	-1,76	[-0,19; 0,66]	0,26	[1; 6]	1,77
StratifyPotency (SP)	\uparrow			\uparrow	$-$	\downarrow	[0; 1]	0,84	[0; 0]	0	[0; 1]	0,84
UnSetClabjType (main) (UCT)		\uparrow		$-$	\uparrow	$-$	[0; 0]	0	[0,1; 1]	0,57	[0; 0]	0
UnSetClabjType (variant) (UCT(v))		\uparrow		$-$	\uparrow	\downarrow	[0; 0]	0	[0,14; 1]	0,61	[0; 5]	1,53
UnSetFeatureType (UFT)		\uparrow		$-$	\uparrow	\downarrow	[0; 0]	0	[0,07; 0,33]	0,13	[0; 1]	0,13

As the table shows, the experiment confirms the intent of all refactorings except one: *extract clabject type* decreased simplicity instead of augmenting it. In the following, we analyse the reason for this discrepancy as well as the rest of results.

- *Collapse clabject with instances* decreases RE of the model the refactoring is applied to, but increases DS of the lower model. Normally, it also decreases SIM because even if a clabject is deleted, its features become copied to each of its instances.

- *Collapse clabject with type* increases RE of the model where the type resides, as this type is added features coming from the clabject. The lower model typically decreases DS as untyped features in the clabject are moved to the upper level; anyhow, the variation interval shows that it may also increase DS depending on the ratio of typed/untyped features in the clabject. SIM increases because, overall, a clabject and its features are deleted.
- *Collapse inheritance hierarchy* does not affect DS because this refactoring is intra-level. If the collapsed clabject has only one direct subclabject, SIM increases and RE decreases because the refactoring deletes one clabject and preserves the number of features. However, if the clabject has several subclabjects, the features of the clabject need to be replicated in all its subclabjects, which makes SIM decrease and RE increase (the number of features becomes larger).
- *Create clabject type* and *create feature type* increase RE (they create a new type for a clabject or feature) and decrease DS (they type an untyped element). The overall size of the multi-level model increases because of the new type, hence SIM decreases.
- *Extract clabject type* creates a type for a set of untyped clabjects, with all fields they have in common. Hence, it increases RE and decreases DS. While it also intends to improve SIM, the experiment reports a slight average increase in overall size. In particular, the expected size reduction is proportional to the number of fields with primitive data type that the untyped clabjects share. However, many refactored clabjects in our evaluation have a high proportion of references (which are not moved to the new type) and few common fields with primitive data type. In consequence, SIM increased for some models in the experiment, but decreased in average.
- *Pull up clabject* and *pull up field* promote a clabject or a field to an upper meta-level. Both increase RE and decrease DS (in the case of pull up clabject, when the clabject is untyped), leaving SIM unaffected.
- *Push down clabject* moves a clabject to the lower meta-level. This decreases RE and, in case the clabject has no ancestors, it augments DS. It may decrease SIM slightly if the refactoring needs to create clabject representatives (see Figure 25).
- *Push down feature* moves a feature from a type to its instances, hence decreasing RE and increasing DS. It may decrease SIM because, in order to move the feature to all instances, representatives may need to be created (see Figure 11).
- *Replace reference by instantiation* converts an occurrence of the type-object pattern into a multi-level solution by promoting some elements one level up. This increases RE of the top model and overall SIM. Moreover, the model containing the reference decreases DS as this reference is replaced by the built-in instantiation relation.
- *Set clabject type* and *set field type* decrease DS and leave RE intact. In addition, *set field type* may increase SIM because its associated indicator counts the number of user-visible fields, which may decrease when a field is assigned a primitive data type.
- *Split clabject* reduces RE as it moves features of a clabject one level down. Since the refactoring also creates new typed elements, DS increases and SIM decreases.
- *Stratify potency* splits a clabject and organizes its fields according to their potency. This increases RE and decreases SIM.
- *Unset clabject type* and *unset feature type* increase DS of the refactored model. In addition, *unset feature type* decreases SIM when the feature has a primitive data type, as the feature becomes visible to the user. The variant of *unset clabject type* has a similar effect, as the clabject keeps the fields of the type, and these fields become visible to the user.

As a summary, Figure 18 represents refactorings in three axes according to their qualitative impact on the three considered quality attributes. Each axis represents a quality attribute (RE, DS, SIM) with three possible values (\uparrow , \cong , \downarrow). For simplicity, the

graphic reflects the average of the different metrics, though some refactorings may produce a result different from the average. For example, *extract clabject type* (ECT) may increase or decrease SIM depending on the number of fields shared by the refactored clabjects.

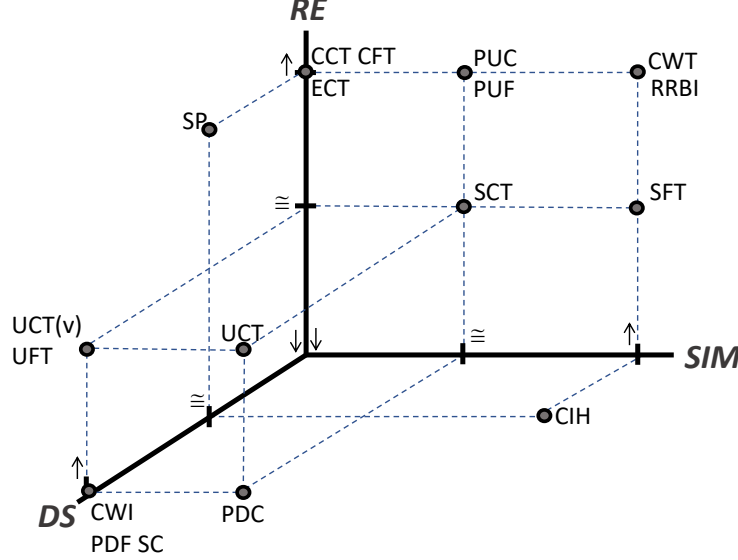


Fig. 18: Tridimensional classification of refactorings according to their impact. The extended name of refactorings can be consulted in Table V

The graphic shows that some refactorings have the same effect on the quality attributes, like PUC and PUF; CCT, CFT and ECT; UCT(v) and UFT; and CWI, PDF and SC. All bottom-up refactorings decrease DS, and most of them (7 out of 9) increase RE. Conversely, all top-down refactorings increase DS, 4 out of 7 decrease RE, and the remaining 3 do not modify RE. Intra-level refactorings (CIH and SP) leave DS intact.

The graphic also reflects the effects of inverse refactorings. For example, PUC increases RE and decreases DS, while PDC has the reverse effect as it decreases RE and increases DS. Other inverse refactorings give reverse results in only one metric, like SCT and UCT which preserve SIM and RE but have opposite effects in DS. Finally, the graphic is useful to identify uncharted space. For example, no refactoring increases both RE and DS. This is an unusual combination as it implies increasing both the size of the top-level model and the number of linguistic extensions at the lower-level model.

7.2.4. Threats to validity. How to measure multi-level models and relate their internal metrics to external ones is currently under debate in the multi-level community [Atkinson and Kühne 2017]. Similarly to previous researchers [Atkinson and Kühne 2008; Gerbig 2017], we chose RE, DS and SIM as quality attributes, and used top-model size and overall size as indicators for RE and SIM. However, other means to assess external attributes like SIM can be appropriate, like user-perceived complexity. Similarly, we calculate DS as the ratio of linguistic extensions with respect to the model size, though additional assessments of DS could be performed by real users. Such assessments would require empirical studies, which we will carry out in the future.

While [Bansiya and Davis 2002] only measures the number of classes, we count clabjects and features because both play an important role in conceptual models. For

simplicity of presentation, the evaluation shows both data aggregated. Anyhow, we manually checked that the aggregated and non-aggregated results vary in the same direction (i.e., we have not introduced errors when simplifying the presentation of results).

In [Bansiya and Davis 2002], the number of hierarchies is used to measure functionality, which can be an additional indicator of RE. To assess whether it may lead to contradictory results with size, we also measured the variation of inheritance hierarchies, noting a positive correlation with size. However, we decided to discard this second indicator as most refactorings in our catalogue do not modify it. Results on the variation of inheritance hierarchies can be found at <http://miso.es/ml-refacts>.

Our refactorings are applied automatically, with no heuristics for their application. Hence, unlike other works in refactorings for class diagrams or object-oriented programs (e.g., [Ghannem et al. 2016; Bavota et al. 2010; Bavota et al. 2011]) our evaluation does not measure the efficacy of any heuristic for refactoring recommendation, but it analyses the impact of refactorings on three quality attributes. Designing heuristics to recommend refactorings depending on metrics like SIM is future work.

8. RELATED WORK

In this section, we revise related works on model refactorings or operations involving several meta-levels (Section 8.1), refactorings working within a single level (Section 8.2), approaches to proving refactoring correctness (Section 8.3), and proposals to automate the detection of refactoring opportunities (Section 8.4).

8.1. Inter-level refactorings

While refactorings have been extensively studied for programming languages and flat models [Fowler 1999; Opdyke and Johnson 1990], they have not been proposed for multi-level models up to now. The closest work to ours is [Atkinson et al. 2012], which explores the use of *emendation services* to repair a multi-level model upon changing some of its elements (e.g., modifying the potency of a clabject or adding an attribute). Our refactorings go beyond atomic changes as they may involve a sequence of modifications, and have a purpose of use, pre-conditions and variants. Emendation services could be used to implement our refactorings in Melanee, which is the technology used in [Atkinson et al. 2012].

Our refactorings are not specific to METADEPTH, but they are applicable to any tool supporting potency-based modelling like Melanee [Atkinson et al. 2012], the extension of Xmodeller described in [Frank 2014], or the ConceptBase implementation in [Neumayr et al. 2014], among others.

In our previous work [de Lara et al. 2014; Macías et al. 2017], we provided some guidelines to rearchitect two-level into multi-level solutions; now, we provide a concrete refactoring (*replace reference by instantiation*) implementing such guidelines, though we impose pre-conditions that were not present in [de Lara et al. 2014] (e.g., clabjects with role *type* cannot participate in references different from the refactored one). In [Jahn et al. 2014], the authors discuss the benefits of allowing inheritance at the instance level, and propose an operator *extract prototype* which extracts the common features of a set of objects, and creates a prototype object from which the other ones inherit. Our refactoring *extract clabject type* is similar, but we create a clabject at a higher level and relate it to the initial clabjects by instantiation instead of inheritance.

Some limited versions of multi-level refactorings can also occur in a two-level setting. For example, promotion transformations [de Lara et al. 2014] can emulate multi-level modelling by converting a model into a meta-model, i.e., they transform each object in the model into a class in the meta-model. Refactoring *replace reference by instantiation* has shown how to refactor a powertype [Odell 1994] into a multi-level model, and its

connection with the type-object pattern [Martin et al. 1997]. Converting an object into a class is similar to our refactoring *pull up clabject*. Bottom-up meta-modelling [López-Fernández et al. 2015] uses an induction algorithm to generalize a set of example models into a meta-model; this algorithm can be implemented using the *extract clabject type* and *extract feature type* bottom-up refactorings. In [Noy and Klein 2004], an ontology-change operation splits a class into several ones and can specify which of the new classes the instances of the old one belong to, thus performing changes in two levels.

Our catalogue includes refactorings that may collapse levels. Some researchers have also proposed mechanisms to flatten multiple levels for different purposes. In [Kainz et al. 2011], promotion transformations are generated by flattening various levels, with the goal of enabling multi-level modelling atop standard two-level meta-modelling tools. In [Guerra and de Lara 2017], the goal of collapsing levels is being able to analyse multi-level integrity constraints by using standard model finders, which only work with two levels at a time. In [Gogolla et al. 2005], a collection of models at different levels is represented in a single model with the aim of comparing several meta-modelling concepts.

Finally, refactorings with side effects – like *push down feature* – may require repairing model instances at lower levels, for example removing slots. This relates to techniques in meta-model/model co-evolution [Cicchetti et al. 2008; Burger and Gruschko 2010], as in a two-level setting, modifying a meta-model may invalidate its existing instances. According to [Cicchetti et al. 2008], model changes can be non-breaking (e.g., generalizing a property), breaking and resolvable (e.g., removing a property) or breaking and non-resolvable (e.g., adding a mandatory class). Hence, our refactorings perform either non-breaking changes, or breaking and resolvable when instances need to be repaired (e.g., when applying *push down feature*). Although some authors [von Pilgrim et al. 2013] have tackled the co-refactoring of models and code, which requires propagating changes between them, such changes cannot be considered to work at two meta-levels because the code is a realization of the design and the authors do not refactor running instances. In [von Pilgrim et al. 2013], refactorings are defined as declarative constraints, which makes them easy to specify but expensive to execute by the use of constraint solving (10 seconds for models of hundreds of elements).

8.2. Intra-level refactorings

Most existing refactoring proposals work on flat models and involve a single level. These refactorings are level-agnostic and could be applied to each level in a multi-level model. Instead, our refactorings are specific to multi-level models and use typical multi-level concepts like potency.

As an example of refactorings applicable within a single level, Sunyé et al propose refactorings for UML class diagrams and state machines [Sunyé et al. 2001], and express their pre- and post-conditions using OCL constraints. Other approaches aim at detecting complex changes in evolved (meta-)models [Hebig et al. 2017]; instead, we assume a library of defined refactorings. Steimann [Steimann 2015] defines constraint-based refactorings where invariants are translated into a constraint satisfaction problem that is used to calculate additional changes for the refactoring to be executable. In [Sun et al. 2013], the authors translate class diagrams (with operation pre- and post-conditions) and sequence diagrams into Alloy, to check whether a particular refactoring application preserves the net effect of the operations. Instead, we have identified pre-conditions that refactorings need to meet, as well as side effects that some of them may produce.

Another aspect related to model-based refactoring is the language used to specify refactorings. Some examples are Epsilon [Arcelli et al. 2018] or different model transformation languages like ATL, Kermeta or QVT [Kolahdouz-Rahimi et al. 2014]. While METADEPTH integrates Epsilon, we opted by a native implementation of the refactor-

ings in Java because this provides more flexibility to encode complex model navigations and element creations in the multi-level models, and to obtain a better integration with the tool. Using Epsilon would have brought extensibility by enabling the addition of user-defined refactorings, but at the price of increased execution time. To promote extensibility of refactorings, we plan to migrate METADEPTH to a component-based architecture able to profit from extension points.

8.3. Refactoring correctness and behaviour preservation

Some approaches to check refactoring correctness are based on formal proofs [Estler and Wehrheim 2008; Gheyi et al. 2005], either proving that specific applications of refactorings are correct, or developing refactoring laws for particular specification languages. For example, in [Gheyi et al. 2005], refactorings for Alloy are studied by translating Alloy specifications and refactorings into the PVS verification system. While this approach could be applicable in our setting, we would still need to assess that the METADEPTH description in PVS conforms to the behaviour of the real running system. Hence, we opted for performing exhaustive testing as it involves the real system and not a description of it. Other approaches specify refactorings formally – for example using graph transformation – and focus on specific properties like conflicts and dependencies between refactorings [Mens et al. 2007]. Such approaches could be used in the future to analyse inter-dependencies of our refactorings, as in this paper, we are just interested in their definition and individual properties.

On a more practical side, some works use testing to check refactoring reliability. However, creating test programs and oracles manually is complex, tedious, and has the potential to yield incomplete test suites. Hence, some works [Daniel et al. 2007; Soares et al. 2013] automatically generate test programs and some form of oracle, ranging from simple ones (e.g., does the refactoring crash?) to differential testing (i.e., testing the same refactoring in different tools) and specialized tools like SafeRefactor [Soares et al. 2010].

Some researchers have investigated ways to detect too strong pre-conditions in Java refactorings. The approach in [Mongiovi et al. 2017] detects situations where a refactoring would be rejected by a pre-condition by applying the refactoring and comparing both behaviours using testing. If the behaviour is the same, a case of overly strong pre-conditions has been found. While this approach is applicable in our setting, investigating overly strong pre-conditions is out of the scope of this paper.

Differential pre-condition checking [Overbey et al. 2016] facilitates pre-condition specification by describing the changes allowed or forbidden by a refactoring on a semantic model of the program. Then, a refactoring application needs to build such a semantic model, simulate the transformation, and compare the actual differences with the specified ones. If the differences match, the refactoring is considered behaviour preserving and can be executed. While this is a minimalistic way to define refactoring pre-conditions, in our setting, it is an open question which semantic model could be powerful enough to express and check all refactoring pre-conditions efficiently. Instead, we resort to a traditional way to encode pre-conditions, while we have assessed semantics preservation based on exhaustive bounded instantiation.

Most approaches checking behaviour preservation of refactorings work at the code-level. One of the few exceptions is [Mansoor et al. 2017], which works with class and activity diagrams. The authors use multi-objective optimization to find the best sequence of refactorings that provide a good trade-off between maximizing the quality of class diagrams and activity diagrams, while preserving the behavioural constraints defined on activity diagrams. As our models do not have a specification of behaviour (e.g., operations), we are concerned with the preservation of the set of instances of the refactored model.

Regarding input program generation, in [Soares et al. 2013; Mongiovi et al. 2017], the authors use Alloy and a Java meta-model to exhaustively generate programs within a given scope. In contrast, [Daniel et al. 2007] follows an imperative approach, proposing a framework that creates generators that synthesize abstract syntax trees which can be combined to produce an exhaustive set of input programs to the refactoring. Both works discovered dozens of bugs in commercial tools by analysing around 150.000 refactoring applications in the biggest experiment (for a set of 29 refactorings).

In our case, using a purely declarative way to specify model shapes, and a constraint solver to generate test models, would be complex. Instead, it is more direct to specify seed models that satisfy the refactoring pre-conditions, and then use mutation operators exhaustively. To the best of our knowledge, this is a novel technique to test model refactorings. Regarding experiment size, we performed more than 210.000 refactoring applications (for a set of 18 refactorings).

8.4. Automated detection of refactoring opportunities

The present paper does not deal with the issue of automatically detecting refactoring opportunities, but users must select the elements to which a refactoring is to be applied. We plan to tackle the partial automation of refactoring recommendation in the future, e.g., by proposing smells that indicate the need for applying a certain refactoring. Next, we comment on some relevant works on this topic.

Some researchers detect model refactoring opportunities by looking for occurrences of anti-patterns. As an example, the anti-patterns in [Arcelli et al. 2018] detect performance smells in UML models, and trigger refactorings targeted to solve the smells. The authors distinguish three refactoring application modes: batch, user-driven multiple (the system detects all anti-patterns, and the user selects the refactoring to apply), and user-driven single (the user selects an element, the system detects all anti-patterns where the element participates, and the user selects the refactoring to apply). In our context, the user-driven multiple mode is the most appropriate.

Other works are based on search. For example, Ghannem et al. [Ghannem et al. 2016] identify refactoring opportunities on class diagrams by using a genetic algorithm based on the similarity between the system under study and a set of defect examples. In [Ouni et al. 2012], the authors look for sequences of refactorings that fix a problem without changing the design semantics, where *semantics* is interpreted as cohesion between class elements. To this aim, they use multi-objective search based on a genetic algorithm that uses criteria like the similarity of vocabulary of code elements. In our case, the semantics is the set of instances of a model. Moreover, complex vocabulary semantic relations – like hypernymy and hyponymy – would be useful to detect refactoring possibilities, in the style of [Bunk et al. 2017].

Obtaining good cohesion is a concern when the standard refactoring *extract class* has to automatically select the most related methods and features to be split into two classes. This is done by analysing the cohesion between each method pair in [Bavota et al. 2010], and considering a metric that combines structure (shared attributes, method calls) and “semantics” (a text similarity metric for identifiers and comments) in [Bavota et al. 2011]. Then, some cohesion metric is used to validate that the generated class split is good. In our case, depending on the refactoring, the features to move out of a clabject can be selected either manually by the user (like in *pull up feature*) or automatically based on their potency (like in *split clabject* and *stratify potency*). This way, when some of our refactorings chooses to move certain features, it does so with the specific intent of organizing them according to their potency. Hence, validating the suitability of the selected feature set does not apply in our case.

9. CONCLUSIONS AND FUTURE WORK

Multi-level modelling is a promising paradigm that promotes flexibility in modelling and yields simpler solutions than two-level approaches in some scenarios [de Lara et al. 2014]. However, it requires flexible exploratory phases of design where information can be rearranged across levels. As manual rearrangements are costly, we have proposed their automated support. Thus, we have analysed and classified useful multi-level refactorings, presenting an initial catalogue and tool support. Moreover, we have validated the side effects of refactorings by performing exhaustive testing using more than 210.000 refactoring applications, verified using more than 20.500.000 instances, and we have evaluated their impact on three quality properties of multi-level models: reusability, simplicity and domain-specificity.

Our refactorings apply to multi-level models, which means they include the “model type” definition at the top-most level. In this sense, they are applicable to any “type of model”. More importantly, our refactorings enhance flexibility in modelling, which is liberated from a “top-down” approach (first types and then instances) but can also proceed “bottom-up” (first instances and then types). Moreover, some of them help in rearchitecting two-level solutions into multi-level ones. To the best of our knowledge, this is the first catalogue of multi-level refactorings proposed in the literature, and we hope it can serve as a basis for their introduction in existing multi-level modelling tools. The approach we took to evaluate the refactorings is also novel and can be used to evaluate refactorings and in-place model transformations for flat models.

In the future, we plan to develop a recommender that detects “bad smells” and suggests refactorings, in the style of [Ouni et al. 2016]. We would also like to devise a language to describe complex refactorings, allowing their concatenation or iteration. We also plan to make the catalogue extensible by rearchitecting METADEPTH to a component architecture (e.g., based on OSGi) where new refactorings can be contributed in a modular way.

Acknowledgements. We thank the anonymous reviewers for their suggestions and comments, which helped improving previous versions of this article. This work has been funded by the Spanish Ministry of Economy and Competitiveness with project “Flexor” (TIN2014-52129-R), and the region of Madrid with project “SICOMORO-CM” (S2013/ICE-3006).

REFERENCES

- João Paulo A. Almeida, Ulrich Frank, and Thomas Kühne. 2017. Multi-level modelling (Dagstuhl seminar 17492). *Dagstuhl Reports* 7, 12 (2017), 18–49. See also <http://www.dagstuhl.de/en/program/calendar/semhp/?seminr=17492>.
- Davide Arcelli, Vittorio Cortellessa, and Daniele Di Pompeo. 2018. Performance-driven software model refactoring. *Information and Software Technology* 95 (2018), 366–397.
- Colin Atkinson, Ralph Gerbig, and Mathias Fritzsche. 2015. A multi-level approach to modeling language extension in the enterprise systems domain. *Inf. Syst.* 54 (2015), 289–307.
- Colin Atkinson, Ralph Gerbig, and Bastian Kennel. 2012. On-the-fly emendation of multi-level models. In *ECMFA (LNCS)*, Vol. 7349. Springer, Berlin, Heidelberg, 194–209.
- Colin Atkinson, Bastian Kennel, and Björn Goß. 2011. Supporting constructive and exploratory modes of modeling in multi-level ontologies. In *SWESE*, 1–15.
- Colin Atkinson and Thomas Kühne. 2002. Rearchitecting the UML infrastructure. *ACM Trans. Model. Comput. Simul.* 12, 4 (2002), 290–321.
- Colin Atkinson and Thomas Kühne. 2008. Reducing accidental complexity in domain models. *Soft. and Syst. Model.* 7, 3 (2008), 345–359.
- Colin Atkinson and Thomas Kühne. 2017. On evaluating multi-level modeling. In *MODELS Satellite Events (CEUR Workshop Proceedings)*, Vol. 2019. 274–277.
- Jagdish Bansiya and Carl G. Davis. 2002. A hierarchical model for object-oriented design quality assessment. *IEEE Transactions on Software Engineering* 28, 1 (2002), 4–17.

- Gabriele Bavota, Andrea De Lucia, Andrian Marcus, and Rocco Oliveto. 2010. A two-step technique for extract class refactoring. In *ASE*. ACM, New York, NY, USA, 151–154.
- Gabriele Bavota, Andrea De Lucia, and Rocco Oliveto. 2011. Identifying extract class refactoring opportunities using structural and semantic cohesion measures. *Journal of Systems and Software* 84, 3 (2011), 397–414.
- Jean Bézivin and Olivier Gerbé. 2001. Towards a precise definition of the OMG/MDA framework. In *ASE*. IEEE Computer Society, Washington, DC, USA, 273–280.
- Marco Brambilla, Jordi Cabot, and Manuel Wimmer. 2012. *Model-Driven Software Engineering in Practice*. Morgan & Claypool, USA.
- Stefan Bunk, Fabian Pittke, and Jan Mendling. 2017. *Aligning process model terminology with hypernym relations*. Springer International Publishing, Cham, 105–123.
- Erik Burger and Boris Gruschko. 2010. A change metamodel for the evolution of MOF-based metamodels. In *Modellierung (LNI)*, Vol. 161. GI, 285–300.
- Antonio Cicchetti, Davide Di Ruscio, Romina Eramo, and Alfonso Pierantonio. 2008. Automating co-evolution in model-driven engineering. In *EDOC*. IEEE Computer Society, Washington, DC, USA, 222–231.
- Tony Clark, Paul Sammut, and James S. Willans. 2015. Super-languages: Developing languages and applications with XMF (Second Edition). *CoRR* abs/1506.03363 (2015). <http://arxiv.org/abs/1506.03363>
- Brett Daniel, Danny Dig, Kely Garcia, and Darko Marinov. 2007. Automated testing of refactoring engines. In *ESEC-FSE*. ACM, New York, NY, USA, 185–194.
- Victorio Albani de Carvalho and João Paulo A. Almeida. 2018. Toward a well-founded theory for multi-level conceptual modeling. *Soft. and Syst. Model.* 17, 1 (2018), 205–231.
- Juan de Lara and Esther Guerra. 2010. Deep meta-modelling with METADEPTH. In *TOOLS (LNCS)*, Vol. 6141. Springer, Berlin, Heidelberg, 1–20. See also <http://metaDepth.org>.
- Juan de Lara and Esther Guerra. 2017. *A posteriori* typing for model-driven engineering: concepts, analysis, and applications. *ACM Trans. Softw. Eng. Methodol.* 25, 4 (2017), 31:1–31:60.
- Juan de Lara, Esther Guerra, Ruth Cobos, and Jaime Moreno-Llorena. 2014. Extending deep meta-modelling for practical model-driven engineering. *Comput. J.* 57, 1 (2014), 36–58.
- Juan de Lara, Esther Guerra, and Jesús Sánchez Cuadrado. 2014. When and how to use multilevel modelling. *ACM Trans. Softw. Eng. Methodol.* 24, 2 (2014), 12:1–12:46. DOI : <http://dx.doi.org/10.1145/2685615>
- Juan de Lara, Esther Guerra, and Jesús Sánchez Cuadrado. 2015. Model-driven engineering with domain-specific meta-modelling languages. *Soft. and Syst. Model.* 14, 1 (2015), 429–459.
- H.-Christian Estler and Heike Wehrheim. 2008. Alloy as a refactoring checker? *Electr. Notes Theor. Comput. Sci.* 214 (2008), 331–357.
- Walid Fdhila, Conrad Indiono, Stefanie Rinderle-Ma, and Manfred Reichert. 2015. Dealing with change in process choreographies: Design and implementation of propagation algorithms. *Inf. Syst.* 49 (2015), 1–24.
- Claudenir Fonseca. 2017. *ML2: An expressive multi-level modeling conceptual modeling language*. Master's thesis. Universidade Federal do Espírito Santo, Brazil.
- Martin Fowler. 1999. *Refactoring - Improving the Design of Existing Code*. Addison-Wesley.
- Ulrich Frank. 2014. Multilevel modeling - Toward a new paradigm of conceptual modeling and information systems design. *Bus. & Inf. Syst. Eng.* 6, 6 (2014), 319–337.
- Ralph Gerbig. 2017. *Deep, seamless, multi-format, multi-notation definition and use of domain-specific languages*. Ph.D. Dissertation. University of Mannheim.
- Adnane Ghannem, Ghizlane El Boussaidi, and Marouane Kessentini. 2016. On the use of design defect examples to detect model refactoring opportunities. *Software Quality Journal* 24, 4 (2016), 947–965.
- Rohit Gheyi, Tiago Massoni, and Paulo Borba. 2005. A rigorous approach for proving model refactorings. In *ASE*. ACM, New York, NY, USA, 372–375.
- Martin Gogolla, Jean-Marie Favre, and Fabian Büttner. 2005. On squeezing M0, M1, M2, and M3 into a single object diagram. In *Workshop on Tool Support for OCL and Related Formalisms (LGL-REPORT-2005-001, EPFL (Switzerland))*.
- Gerd Gröner, Fernando Silva Parreiras, and Steffen Staab. 2010. Semantic recognition of ontology refactoring. In *ISWC (LNCS)*, Vol. 6496. Springer, Berlin, Heidelberg, 273–288.
- Nicola Guarino. 2004. Toward a formal evaluation of ontology quality. *IEEE Intelligent Systems* 19 (2004), 78–80.
- Esther Guerra and Juan de Lara. 2017. Automated analysis of integrity constraints in multi-level models. *Data Knowl. Eng.* 107 (2017), 1–23.
- Regina Hebig, Djamel Eddine Khelladi, and Reda Bendraou. 2017. Approaches to co-evolution of metamodels and models: A survey. *IEEE Transactions on Software Engineering* 43, 5 (2017), 396–414.

- Muzaffar Igamberdiev, Georg Grossmann, Matt Selway, and Markus Stumptner. 2018. An integrated multi-level modeling approach for industrial-scale data interoperability. *Soft. and Syst. Model.* 17, 1 (2018), 269–294.
- Daniel Jackson. 2002. Alloy: a lightweight object modelling notation. *ACM Trans. Softw. Eng. Methodol.* 11, 2 (2002), 256–290.
- Matthias Jahn, Bastian Roth, and Stefan Jablonski. 2014. Instance specialization - a pattern for multi-level metamodelling. In *MULTI@MODELS (CEUR)*, Vol. 1286. CEUR-WS.org, 23–32.
- Manfred A. Jeusfeld and Bernd Neumayr. 2016. DeepTelos: Multi-level modeling with most general instances. In *ER (LNCS)*, Vol. 9974. Springer International Publishing, Cham, 198–211.
- Gerd Kainz, Christian Buckl, and Alois Knoll. 2011. Automated model-to-metamodel transformations based on the concepts of deep instantiation. In *MODELS (LNCS)*, Vol. 6981. Springer, Berlin, Heidelberg, 17–31.
- S. Kolahdouz-Rahimi, K. Lano, S. Pillay, J. Troya, and P. Van Gorp. 2014. Evaluation of model transformation approaches for model refactoring. *Science of Computer Programming* 85 (2014), 5–40.
- Thomas Kühne and Daniel Schreiber. 2007. Can programming be liberated from the two-level style? – Multi-level programming with DeepJava. In *OOPSLA*. ACM, New York, NY, USA, 229–244.
- Yngve Lamo, Xiaoliang Wang, Florian Mantz, Øyvind Bech, Anders Sandven, and Adrian Rutle. 2013. DPF Workbench: a multi-level language workbench for MDE. *Proceedings of the Estonian Academy of Sciences* 62, 1 (2013), 3–15.
- Jesús J. López-Fernández, Jesús Sánchez Cuadrado, Esther Guerra, and Juan de Lara. 2015. Example-driven meta-model development. *Soft. and Syst. Model.* 14, 4 (2015), 1323–1347.
- Fernando Macías, Esther Guerra, and Juan de Lara. 2017. Towards rearchitecting meta-models into multi-level models. In *ER (LNCS)*, Vol. 10650. Springer, 59–68.
- Fernando Macías, Adrian Rutle, and Volker Stolz. 2016. MultEcore: Combining the best of fixed-level and multilevel metamodelling. In *MULTI@MODELS (CEUR Workshop Proceedings)*, Vol. 1722. CEUR-WS.org, 66–75.
- Usman Mansoor, Marouane Kessentini, Manuel Wimmer, and Kalyanmoy Deb. 2017. Multi-view refactoring of class and activity diagrams using a multi-objective evolutionary algorithm. *Software Quality Journal* 25, 2 (2017), 473–501.
- Robert C. Martin, Dirk Riehle, and Frank Buschmann. 1997. *Pattern Languages of Program Design 3*. Addison-Wesley.
- Tom Mens, Gabriele Taentzer, and Olga Runge. 2007. Analysing refactoring dependencies using graph transformation. *Soft. and Syst. Model.* 6, 3 (2007), 269–285.
- Tom Mens and Tom Tourwé. 2004. A survey of software refactoring. *IEEE Transactions on Software Engineering* 30, 2 (2004), 126–139.
- M. Mongiovi, R. Gheyi, G. Soares, M. Ribeiro, P. Borba, and L. Teixeira. 2017. Detecting overly strong preconditions in refactoring engines. *IEEE Transactions on Software Engineering* in press, 99 (2017), 1–1.
- Bernd Neumayr, Manfred A. Jeusfeld, Michael Schrefl, and Christoph G. Schütz. 2014. Dual deep instantiation and its ConceptBase implementation. In *CAiSE (LNCS)*, Vol. 8484. Springer, Cham, 503–517.
- Bernd Neumayr, Christoph G. Schuetz, Christian Horner, and Michael Schrefl. 2017. DeepRuby: Extending Ruby with dual deep instantiation. In *MODELS Satellite Events (CEUR Workshop Proceedings)*, Vol. 2019. CEUR-WS.org, 252–260.
- Bernd Neumayr, Christoph G. Schuetz, Manfred A. Jeusfeld, and Michael Schrefl. 2018. Dual deep modeling: multi-level modeling with dual potencies and its formalization in F-Logic. *Soft. and Syst. Model.* 17, 1 (2018), 233–268.
- James Noble, Antero Taivalsaari, and Ivan Moore. 1999. *Prototype-based programming: concepts, languages and applications*. Springer.
- Natalya Fridman Noy and Michel C. A. Klein. 2004. Ontology evolution: Not the same as schema evolution. *Knowl. Inf. Syst.* 6, 4 (2004), 428–440.
- James Odell. 1994. Power types. *JOOP* 7, 2 (1994), 8–12.
- OMG. 2016. MOF 2.5.1. <http://www.omg.org/spec/MOF/2.5.1/>. (2016).
- William F. Opdyke and Ralph E. Johnson. 1990. Refactoring: An aid in designing application frameworks and evolving object-oriented systems. In *SOOPPA*. ACM.
- A. Ouni, M. Kessentini, H. Sahraoui, and M. S. Hamdi. 2012. Search-based refactoring: Towards semantics preservation. In *ICSM*. IEEE Computer Society, Washington, DC, USA, 347–356.

- Ali Ouni, Marouane Kessentini, Houari Sahraoui, Katsuro Inoue, and Kalyanmoy Deb. 2016. Multi-criteria code refactoring using search-based software engineering: An industrial case study. *ACM Trans. Softw. Eng. Methodol.* 25, 3, Article 23 (2016), 53 pages.
- Jeffrey L. Overbey, Ralph E. Johnson, and Munawar Hafiz. 2016. Differential precondition checking: a language-independent, reusable analysis for refactoring engines. *Automated Software Engineering* 23, 1 (2016), 77–104.
- Richard Paige, Dimitrios Kolovos, Louis Rose, Nicholas Drivalos, and Fiona Polack. 2009. The design of a conceptual framework and technical infrastructure for model management language engineering. In *ICECCS*. IEEE Comp. Soc., Washington, DC, USA, 162–171.
- Alessandro Rossini, Juan de Lara, Esther Guerra, and Nikolay Nikolov. 2015. A comparison of two-level and multi-level modelling for cloud-based applications. In *ECMFA (LNCS)*, Vol. 9153. Springer, Cham, 18–32.
- Douglas C. Schmidt. 2006. Guest Editor's Introduction: Model-Driven Engineering. *IEEE Computer* 39, 2 (2006), 25–31. DOI : <http://dx.doi.org/10.1109/MC.2006.58>
- Matt Selway, Markus Stumptner, Wolfgang Mayer, Andreas Jordan, Georg Grossmann, and Michael Schrefl. 2017. A conceptual framework for large-scale ecosystem interoperability and industrial product lifecycles. *Data & Knowledge Engineering* 109 (2017), 85–111.
- Gustavo Soares, Rohit Gheyi, and Tiago Massoni. 2013. Automated behavioral testing of refactoring engines. *IEEE Transactions on Software Engineering* 39, 2 (2013), 147–162.
- Gustavo Soares, Rohit Gheyi, D. Servey, and Tiago Massoni. 2010. Making program refactoring safer. *IEEE Software* 27, 4 (2010), 52–57.
- Friedrich Steimann. 2015. From well-formedness to meaning preservation: model refactoring for almost free. *Soft. and Syst. Model.* 14, 1 (2015), 307–320.
- Dave Steinberg, Frank Budinsky, Marcelo Paternostro, and Ed Merks. 2008. *EMF: Eclipse Modeling Framework, 2nd Edition*. Addison-Wesley Professional, Upper Saddle River, NJ.
- Wuliang Sun, Robert B. France, and Indrakshi Ray. 2013. Analyzing behavioral refactoring of class models. In *Workshop on Models and Evolution @ MODELS (CEUR Workshop Proceedings)*, Vol. 1090. CEUR-WS.org, 70–79.
- Gerson Sunyé, Damien Pollet, Yves Le Traon, and Jean-Marc Jézéquel. 2001. Refactoring UML models. In *UML (LNCS)*, Vol. 2185. Springer, London, UK, 134–148.
- UML 2017. UML 2.5.1 OMG specification. <http://www.omg.org/spec/UML/2.5.1/>. (2017).
- Gregory Vial. 2015. Database refactoring: Lessons from the trenches. *IEEE Software* 32, 6 (2015), 71–79.
- Jens von Pilgrim, Bastian Ulke, Andreas Thies, and Friedrich Steimann. 2013. Model/code co-refactoring: An MDE approach. In *ASE*. IEEE, 682–687.
- Barbara Weber, Manfred Reichert, Jan Mendling, and Hajo A. Reijers. 2011. Refactoring large process model repositories. *Computers in Industry* 62, 5 (2011), 467–486.
- Jonathan Yu, James A. Thom, and Audrey M. Tam. 2009. Requirements-oriented methodology for evaluating ontologies. *Inf. Syst.* 34, 8 (2009), 766–791.
- Olaf Zimmermann. 2015. Architectural refactoring: A task-centric view on software evolution. *IEEE Software* 32, 2 (2015), 26–29.

A. APPENDIX: CATALOGUE OF REFACTORINGS

This appendix details the refactorings not described in detail in the body of the paper.

A.1. Collapse clabject with instances

This refactoring collapses a clabject C with its instances one level below.

Type. Top-down; might be level collapsing; composite; no semantic side effects.

Utility. A concept C deemed general across domains is found to be an overgeneralization, though it still provides a valid characterization of its instances. In such a case, this refactoring moves the information from C to its instances – which become linguistic extensions – and C itself is deleted.

Example. In Figure 19, the designer has realized that ResourceType in model (a) is unnecessary in other domains, but its instances still need to define a date. Hence, the designer applies this refactoring to collapse the clabject with its instances. As a result, the clabject is removed, and its attribute date is added to its instances Environment and Artifact with decreased potency 1, but not to Code as this inherits

the attribute from Artifact. Clabjects Artifact, Environment and Code become linguistic extensions. The top-most level is removed because it becomes empty after deleting ResourceType.

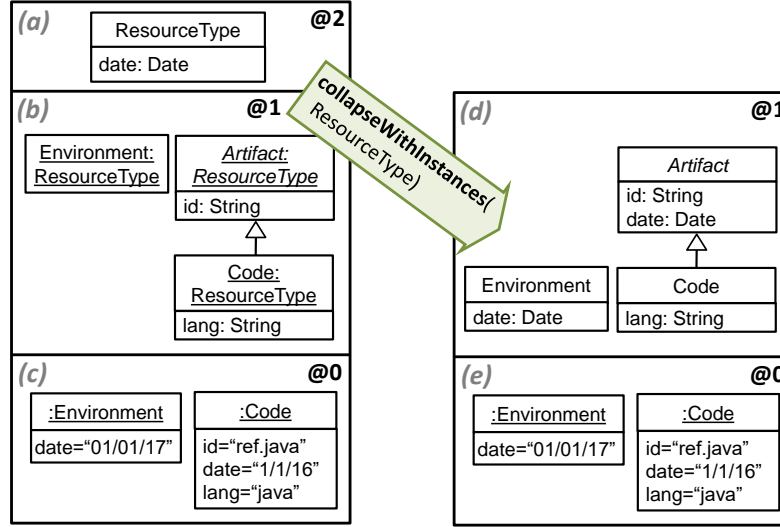


Fig. 19: Example of refactoring *collapse clabject with instances*: ResourceType is collapsed with all its instances (Artifact and Environment)

Consequences. The instances of the collapsed clabject C become linguistic extensions. The features of C are replicated across all its instances with no ancestors. If this replication is not desired, refactoring *push down clabject* (which pushes down a clabject creating a representative at the lower level) should be used instead. The refactoring can be level collapsing if the level that contained C becomes empty.

Pre-conditions. The clabject C to be collapsed cannot have an ontological type, define subclabjects, more than one superclabject, or be the source or target of any reference.

Working scheme. Refactoring *collapseWithInstances(C)*, where C is the clabject to collapse, works as follows:

- (1) Apply refactoring *pushDownClabject(C)*.
- (2) Apply refactoring *collapseHierarchy(C)*.

In a first step, applying refactoring *push down clabject* creates a clabject representative of C in the lower level, and deletes C (see Section A.7). The representative is an abstract clabject from which all instances of C become to inherit, and declares all features of C with decreased potency. In a second step, refactoring *collapse inheritance hierarchy* collapses the clabject representative with its direct subclabjects, which then become owners of the features of the representative (see Section A.3).

A.2. Collapse clabject with type

This refactoring collapses a clabject C with its type, one level above.

Type. Bottom-up; might be level collapsing; atomic; no semantic side effects.

Utility. A clabject C has been created artificially to act as a container of features, which are found to characterize all clabjects of the same kind as C. This means that the features have been created at the wrong meta-level. After collapsing C with its type T, the latter receives C's features with increased potency, becoming available to all

instances of T. This way, the features are generalised. The refactoring is also useful to migrate from a multi-level approach lacking deep characterization (e.g., based on clabject representatives or powertypes) to a potency-based approach (see Figure 13).

Example. In Figure 20, the designer realises that the features of clabject SEArtifact in model (b) apply to resource types in other domains. Since this clabject was created specifically to hold such features, refactoring *collapse clabject with type* is applied. This refactoring generalizes those features by collapsing SEArtifact with its type ResourceType, and moving feature description to ResourceType with potency 2. The instances of subclassjects of SEArtifact remain unaffected (see salesModule at the bottom level). The refactoring is level collapsing, and would remove model (b) should no other clabject existed at that level or below.

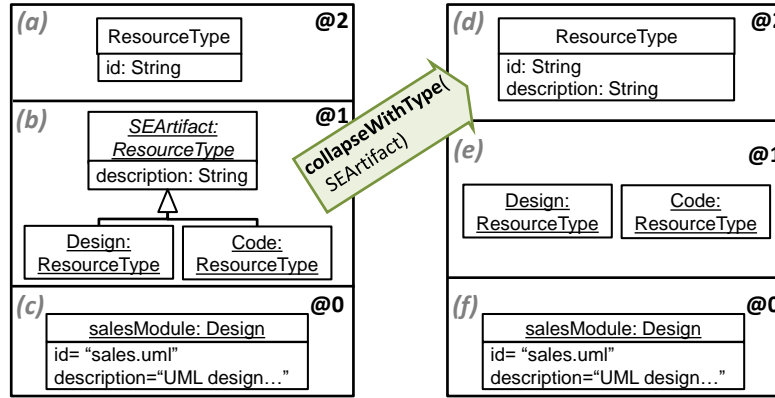


Fig. 20: Example of refactoring *collapse clabject with type*: SEArtifact is collapsed with ResourceType

Consequences. The type the clabject is collapsed to receives new features, which become available to all its instances. The collapsed clabject is removed; if this is not desired, but only its features should be generalized, refactoring *pull up feature* should be used instead (see Section A.6).

Pre-conditions. The clabject C to be collapsed cannot have instances but must have a type. The type of C cannot have other instances apart from C. The target of references owned by C, and the source of references reaching C, should have a type.

Working scheme. Refactoring *collapseWithType(C)*, where C is the clabject to collapse with its type, works as follows:

- (1) For each feature f of C, apply refactoring *pullUpFeature(C.f)*.
- (2) For each incoming reference D.r to C, apply refactoring *pullUpFeature(D.r)*.
- (3) Remove C.
- (4) If the container model of C becomes empty, remove it.

First, refactoring *pull up feature* moves all features of C to its type (step 1), and its incoming references are also moved one level up (step 2). The pulled-up features have increased potency, and in case of references, their lower bound is relaxed to 0. Next, C is deleted (step 3), and if the model that contained C becomes empty, it is deleted as well (step 4).

A.3. Collapse inheritance hierarchy

This refactoring collapses a clabject with its direct subclassjects in the inheritance hierarchy. It is similar to the *collapse hierarchy* refactoring in standard object-oriented

design [Fowler 1999], but considering the potency of the involved elements. Moreover, in the standard refactoring, it is possible to select the class to collapse, either the superclass or the subclass. This is also possible in our setting, but keeping the subclass would restrict our refactoring to situations where the superclabject has only one subclass.

Type. Intra-level; level preserving; atomic; no semantic side effects.

Utility. A superclabject adds little value to its subclabjects, so it is merged with them. A cause of this situation may be the *speculative generality* design smell [Fowler 1999]. In a multi-level context, this smell arises when the instances of a superclabject seldom use its features (i.e., their value is left undefined) or when a superclabject only has a subclass and no further extension is foreseen.

Example. In Figure 21, clabject Resource Type is not deemed necessary because the designer does not foresee any other subclass apart from Software Type. After collapsing both clabjects, Software Type retains its potency and receives the field critical that was originally defined by Resource Type. Please note that clabjects may have the same or higher potency than their superclabjects.

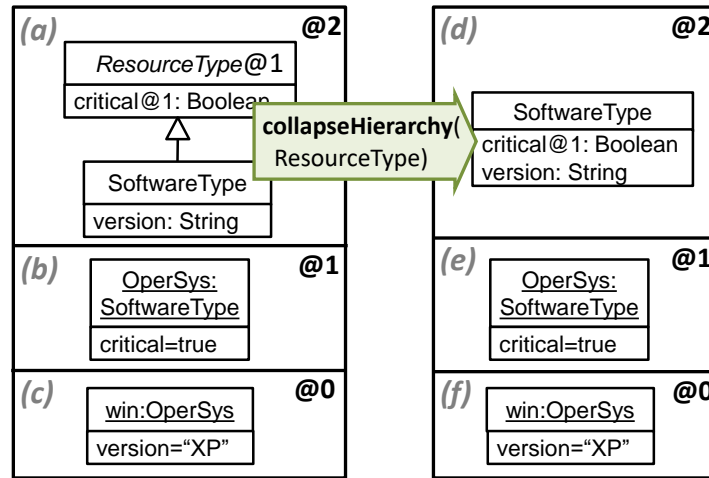


Fig. 21: Example of refactoring *collapse inheritance hierarchy*: Resource Type is collapsed with Software Type

Consequences. The inheritance hierarchy becomes more compact and simpler. The collapsed clabject may receive features with lower potency than the clabject, hence achieving an effect contrary to the *stratify potency* refactoring (Section A.11), which separates features with different potencies across a hierarchy of clabjects. A stratification of features in clabjects according to their potency may promote reusability and yield more flexible models [de Lara et al. 2014].

Pre-conditions. The clabject C to be collapsed cannot have instances or incoming references, and must define at least one subclass.

Working scheme. Refactoring `collapseHierarchy(C)`, where C is the clabject to collapse with its subclabjects, works as follows:

- (1) For each subclabject D of C:
 - (a) add the superclabjects of C as superclabjects of D.
 - (b) remove C as superclabject of D.
 - (c) clone each feature of C in D.
- (2) Delete C.

In this refactoring, each child D of C is added the superclabjects of C (step 1a) and the features of C (step 1c). Clabject C is removed as superclabject of D (step 1b), and C is deleted. As the subclabjects of C can hold a potency equal or higher than the potency of C, they retain their potency.

A.4. Create clabject type

This refactoring creates a new clabject type for an untyped clabject C. If C has subclabjects, these get typed to the newly created clabject type as well.

Type. Bottom-up; might be level expanding; atomic; no semantic side effects.

Utility. A concept that is represented by an untyped clabject in an intermediate meta-level (i.e., a linguistic extension) is relevant at a higher level because either the concept is useful in other domains, or it is recurrent within the same domain (i.e., other instances of the concept need to be created). Hence, this refactoring creates a type for the clabject at the upper meta-level.

Example. In Figure 22, model (b) defines the concept of *resource* for the software engineering domain, but this concept is also useful in other domains. Hence, the refactoring is applied to *SEResource* to make this concept more general. This adds a new clabject *ResourceType* one level above, from which *SEResource* and its child clabject *Code* become instances. The features of *SEResource* (i.e., *date*) and the references where it participates (i.e., *produces* and *consumes*) are not pulled up to *ResourceType*; to obtain that behaviour, refactoring *extract clabject type* should be used instead.

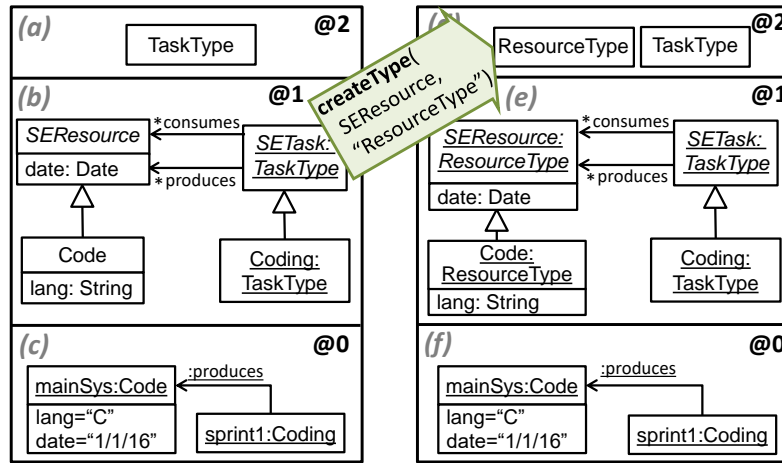


Fig. 22: Example of refactoring *create clabject type*: clabject *ResourceType* is created as type for *SEResource*

Consequences. The newly created type becomes applicable to other domains and can be used to create new instances, which nonetheless have no features. If the features of the clabject used to create the type are deemed generalizable, then refactoring *extract clabject type*, which pulls up all the features as well, should be used instead (see Section 4.2). Alternatively, the type of selected features of the clabject can be created using refactoring *create feature type* (see Section A.5).

Pre-conditions. The clabject C cannot have a type or superclabjects. The upper meta-level cannot have a clabject with same name as the newly created type.

Working scheme. Refactoring createType(C, name), where C is the clobject to type and name its new name, works as follows:

- (1) If C resides in a model M with no ontological type, then create a model MM, and make MM the ontological type of M. This causes a level expansion.
- (2) Create a new non-abstract clobject C' in the ontological type of M.
- (3) Rename C' to name, and give it the potency of C plus 1.
- (4) Set the ontological type of C and its children clobjects to C'.

A.5. Create feature type

This refactoring creates an ontological type for a feature lacking one.

Type. Bottom-up; level preserving; atomic; it has semantic side effects.

Utility. A clobject with ontological type T defines an untyped feature (i.e., a linguistic extension), and this feature is deemed more general and applicable to other instances of T. After applying the refactoring, the untyped feature gets typed by a new feature in T, and this new feature becomes available to all instances of T.

Example. In Figure 23, the designer identifies that reference SETask.consumes (a linguistic extension) should be generalised because it is useful in other domains. For this purpose, the refactoring creates the new reference requires with cardinality 0..* between the ontological types of SETask and SEResource. The new reference becomes the type of SETask.consumes.

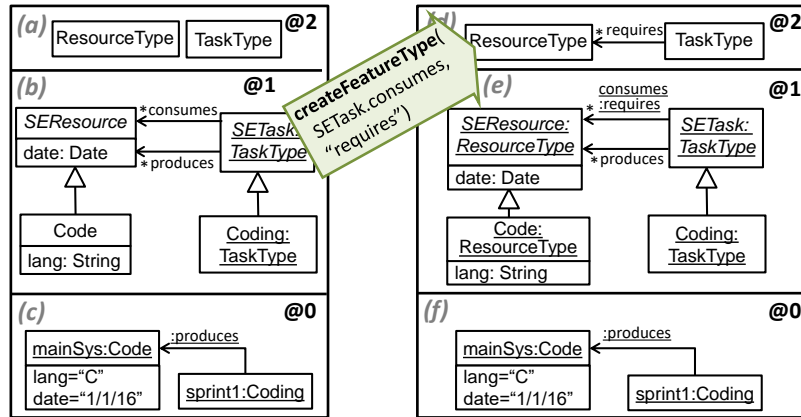


Fig. 23: Example of refactoring *create feature type*: reference *requires* is created as type for *consumes*

Consequences. The ontological type of the clobject that owns the feature f to be typed receives a new feature, which becomes the type of f. If f is a reference, the owner clobject of f can define new links with the same type as f. If f is a field, the refactoring has side effects as existing instances of the ontological type of the owner clobject of f will require a slot for the new field.

Pre-conditions. The feature to be typed should not have a type, but its owner clobject should be typed. If the feature is a reference, its target clobject should be typed as well. The ontological type of the clobject that owns the feature should not have another feature with same name as the newly created feature.

Side effects. This refactoring has side effects when the feature to be typed is a field (not a reference), and there are more clobjects with the same ontological type as the owner of the field. The effect is that these other clobjects receive the new field.

Working scheme. Refactoring `createFeatureType(C.f, name)`, where `f` is an untyped feature owned by `C` and `name` is the name of its new ontological type, works as follows:

- (1) Let `T` be the type of `C`.
- (2) Create a clone `f'` of `f` in `T`, with its potency increased by 1, and name `name`.
- (3) Set the type of `C.f` to `T.f'`.
- (4) If `T.f'` is a reference:
 - (a) set its cardinality to `0..*`.
 - (b) set its target to the ontological type of the target clobject of `C.f`.
- (5) If `T.f'` is a field:
 - (a) set its default value according to its data type.
 - (b) for each clobject `C' ≠ C` s.t. `C'` is a direct or indirect instance of `T` and its potency is in the range `[T.potency-1..T.potency-T.f'.potency]`:
 - i. create an instance of `f` in `C'`.

The algorithm clones the feature `f` in the ontological type `T` of the owner clobject, and this clone becomes the type of `f` (steps 2 and 3). If `f` is a reference, its type receives cardinality `0..*` to increase generality and avoid breaking existing instances (step 4a), and its target is the ontological type of the target of `f` (step 4b). If `f` is a field, its type is assigned an appropriate default value (e.g., 0 in case of integers), and then it is instantiated in all direct and indirect instances of `T` (step 5b). The traversal of the instances of `T` is performed top-down from higher to lower levels to ensure a correct instantiation of the field. This repair action is unnecessary for references, as the minimum cardinality of the reference type is set to 0 (step 4a). Please note that Figure 23 does not show the intermediate field instantiations.

A.6. Pull up feature

This refactoring moves a feature to the ontological type of the owner clobject.

Type. Bottom-up; level preserving; atomic; it has semantic side effects.

Utility. A clobject with ontological type `T` defines an untyped feature (i.e., a linguistic extension), and this feature is deemed more general and applicable to all other instances of `T`. After applying the refactoring, the feature is moved to `T` so that it characterizes all its instances.

Example. In Figure 24, feature `SEResource.date` in model (b) also applies to other types of resource like `ComputingSystem`, and should be generalised. This refactoring does so by moving the feature to `ResourceType` in model (d) with increased potency and a default value. After pulling up the feature, it becomes available to `ComputingSystem` and its instances, hence producing a side effect.

Consequences. The pulled-up feature becomes applicable to any other existing or new instance of the clobject where the feature is moved to. The consequences of this refactoring are similar to those of *create feature type*, but while *create feature type* creates a type for the feature, *pull up feature* in addition removes the feature (i.e., the feature is effectively moved one level above). This difference is only evident in case of references, as fields need to be instantiated at any possible level if their potency allows so, hence their removal is not possible.

Pre-conditions. The feature to be pulled up, and its owner clobject, should have a type. If the feature is a reference, it should have no instances, and its target should have a type.

Side effects. This refactoring has side effects when the feature to be typed is a field (not a reference), and there are more clobjects with the same ontological type as the owner of the field. The effect is that these other clobjects receive the new field.

Working scheme. Refactoring `pullUpFeature(C.f)`, where `f` is an untyped feature owned by `C`, works as follows:

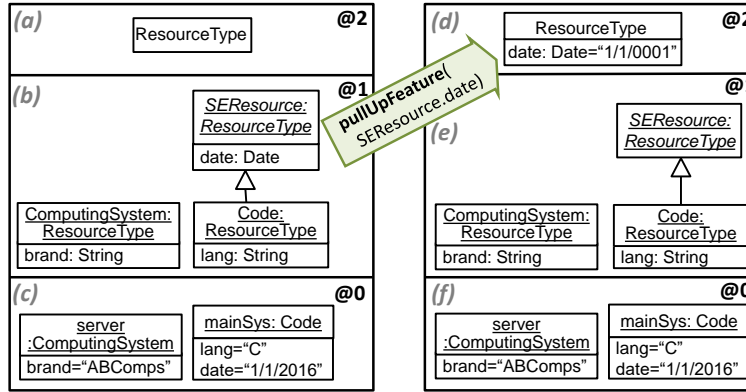


Fig. 24: Example of refactoring *pull up feature*: date is moved from SEResource to ResourceType

- (1) Let T be the type of C .
- (2) Create a clone f' of f in T , with its potency increased by 1.
- (3) If $T.f'$ is a reference:
 - (a) set its minimum cardinality to 0.
 - (b) set its target to the ontological type of the target clabject of $C.f$.
 - (c) remove $C.f$.
- (4) If $T.f'$ is a field:
 - (a) set its default value according to its data type.
 - (b) for each clabject $C' \neq C$ s.t. C' is a direct or indirect instance of T and its potency is in the range $[T.potency-1..T.potency-T.f'.potency]$:
 - i. create an instance of f in C' .
 - (c) set the type of $C.f$ to $T.f'$.

The working scheme is similar to the one for *create feature type*. The feature is cloned in the type of the owner clabject (step 2). If it is a reference (step 3), its cardinality is generalized, its target is set to the type of the target of the original feature, and the feature itself deleted. If it is a field (step 4), it becomes an instance of the created clone, and the latter is instantiated in all direct and indirect instances of its owner clabject. Please note that clabject *server* in model (f) of Figure 24 does not show the feature *date* because we omit the features with a default value that is not overridden.

A.7. Push down clabject

This refactoring moves a clabject and its features one meta-level down.

Type. Top-down; might be level collapsing; atomic; no semantic side effects.

Utility. A clabject needs to be made less general because it is specific to some domain.

The repeated application of this refactoring can be used to migrate a multi-level solution into a two-level one, where instantiation relations are replaced by inheritance. This idiom to encode a multi-level model is called *static types* in [de Lara et al. 2014].

Example. In Figure 25, model (a) defines the clabject *Documentation*, which is a concept used in software engineering but it is not general for arbitrary process modelling domains. The refactoring moves the clabject one level down with decreased potency, and makes it abstract. The instances of *Documentation* (*CodeComments* and *DesignDoc*) get to inherit from it. The features of *Documentation* (*size* and *describes*) are pushed down with decreased potency as well. Moreover, an abstract clabject *SETask* is created as representative of all instances of *TaskType*, in order to use it as the type of reference *Documentation.describes*.

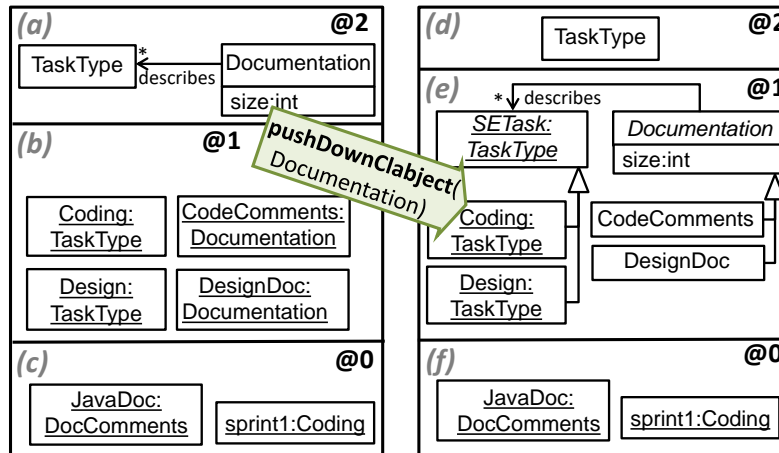


Fig. 25: Example of refactoring *push down clabject*: clabject Documentation is moved from level 2 to level 1

Consequences. This refactoring replaces instantiation by inheritance: the instances of the clabject being pushed down become their subclabjects, and clabject representatives may need to be created (see Section 5.2). If the clabject had instances, it becomes abstract once pushed down. The pushed-down clabject is deleted from its original level; if this is not desired, but the clabject and some of its features should be left in its original level, refactoring *split clabject* should be used instead (see Section A.10).

Pre-conditions. The clabject to be pushed down cannot have a type, be the target of references, or define features with potency 1 or 0. If the clabject inherits features with potency 1, they should be optional or have a default value. If the clabject defines references, their target clabject should have instances, but not the references. The clabject cannot have subclabjects, or more than one superclabject.

Working scheme. Refactoring *pushDownClabject*(C, name?), where C is the clabject to push down, and name is an optional parameter that establishes a new name for the clabject, works as follows:

- (1) Let M be the container model of C, and M' an instance of M.
- (2) Create a clabject C' in M' with the potency of C minus 1.
- (3) Set the name of C' to name if parameter name is given, or to the name of C otherwise.
- (4) If C has a superclabject D, make D the type of C'.
- (5) If C has instances, make C' abstract.
- (6) For each instance I of C:
 - (a) if C has a superclabject D, make D the type of I.
 - (b) make C' the superclabject of I.
- (7) For each feature f in C, apply refactoring *pushDownFeature*(C.f, {C'}).
- (8) Remove C.
- (9) If M becomes empty, remove M.

The first step identifies the model M' where C is to be pushed down. In practice, this is obtained from the context model where the refactoring command is issued. Since M' always exists, the refactoring cannot be level expanding. Then, steps 2 and 3 create a clabject C' in M' with the appropriate name. Step 4 changes inheritance by instantiation (the superclabject of C becomes the type of C'), while step 6 changes

instantiation by inheritance (instances of C become subcljects of C'). Step 7 pushes down each feature of C to C' (see Section 4.3). Since C' is the representative of all instances of C, *push down feature* has no semantic side effects. The last steps remove C, as well as the container model of C if it becomes empty.

A.8. Set clabject type

This refactoring assigns an existing type to an untyped clabject, attempting to type as many features of the clabject as possible.

Type. Bottom-up; level preserving; atomic; it may have semantic side effects.

Utility. A domain-specific concept C (i.e., a linguistic extension) is found to be an ontological instance of an existing clabject T. Hence, the refactoring assigns the type T to C, and types as many features of C as possible with respect to matching features of T. C can be the target of references pointing to instances of T. Any model management operation defined over T becomes applicable to C.

Example. In Figure 26, the designer realizes that the linguistic extension DevEnvironment in model (b) is a kind of resource, and hence, the designer applies the refactoring to set the type of DevEnvironment to ResourceType. As a result, reference DevEnvironment.requiredFor gets typed by reference ResourceType.usedIn. The example shows that the refactoring may have semantic side effects. In this case, clabject Eclipse in the bottom level receives the field date after applying the refactoring, as ResourceType defines this field with potency 2. Should ResourceType.date would have been mandatory (i.e., with lower cardinality bigger than 0), the refactoring would have assigned a default value to field Eclipse.date.

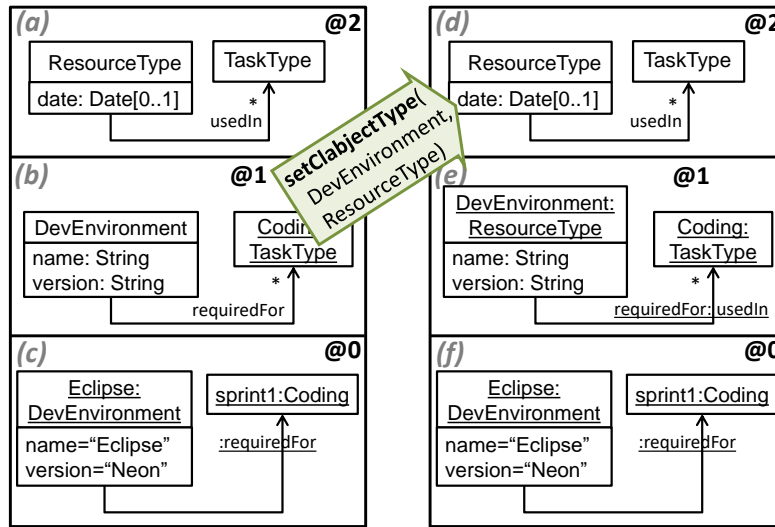


Fig. 26: Example of refactoring *set clabject type*: DevEnvironment gets typed by ResourceType

Consequences. The clabject and its subcljects become typed, as well as the subset of their features matching some in the assigned clabject type. The unmatched features in the clabject remain linguistic extensions. The clabject and its instances are increased with the unmatched fields of the clabject type, which is a side effect of the refactoring.

Pre-conditions. The clabject C to be typed must be untyped, have no superclabjects, and its potency should be one less than the potency of its new type T. If C and T have fields f and f' with primitive data type and same name, their data type must be the same, and the potency of f must be one less than the potency of f'. All mandatory features of T with potency 1 should be matched by some feature in C.

Side effects. There are semantic side effects when the clabject type has unmatched fields after the refactoring (e.g., date in Figure 26).

Working scheme. Refactoring setClabjectType(C, T), where C is the clabject to type by clabject T, works as follows:

- (1) The type of C and all its subclabjects is set to T.
- (2) For each feature f owned by C or its subclabjects:
 - (a) if there is a compatible feature f' in T, apply setFeatureType(f, f').
- (3) For each unmatched primitive field f' of T:
 - (a) create an instance of f' in C and every subclabject.
 - (b) for each instance l of C or its subclabjects:
 - i. create an instance of f' in l.
 - ii. if the created instance has potency 0 and f' is mandatory, assign a default value to the created instance according to its type.

In step 1, the type of C and all its subclabjects is set to T. Then, step 2 assigns a type to as many features of C and its subclabjects as possible, applying refactoring *set feature type* (see Section A.9). A feature f of C matches a feature f' of T if: (i) the potency of f plus 1 equals the potency of f'; (ii) the types of f and f' match; (iii) if f and f' are fields, their names are equal. If the features are fields, condition (ii) means equality of data types; in case of references, it means that the target of f should be typed by the target of f' or a superclabject. Step 3 creates instances of the unmatched fields of T in C, its subclabjects, and their instances. If the unmatched field is mandatory, the created instances with potency 0 receive a value according to the field data type. This last step, which adds the necessary field instances to the clabjects that need to be repaired, is the responsible of the side effects. Please note that Figure 26 does not show intermediate field instantiations (e.g., DevEnvironment.date in model (e)) or uninitialized instantiations of optional fields (e.g., Eclipse.date in model (f)).

A.9. Set feature type

This refactoring assigns an existing type to a feature lacking one.

Type. Bottom-up; level preserving; atomic; no semantic side effects.

Utility. A domain-specific feature f unique for a clabject (i.e., a linguistic extension) is found to be an instance of an existing feature f'. Hence, the refactoring assigns the type f' to f.

Example. In Figure 27, reference SETask.produces in model (b) is a linguistic extension unique to software engineering tasks. As an afterthought, the designer realizes that the reference represents a particular use of a resource, and therefore, it can be recasted as an instance of TaskType.uses. Applying the refactoring sets the type of SETask.produces to TaskType.uses, so that it can be treated uniformly with the rest of instances of the latter. In this way, an expression like self.uses defined on TaskType and evaluated at sprint1 on model (f) returns the union of the instances of references consumes and produces defined in model (e).

Consequences. The feature becomes typed and is no longer a linguistic extension. If it is a reference, this means that it can be treated uniformly with the other instances of the assigned type.

Pre-conditions. The feature f to be typed should be untyped, but its owner clabject C should have a type. The feature type f' assigned to f should be defined in the

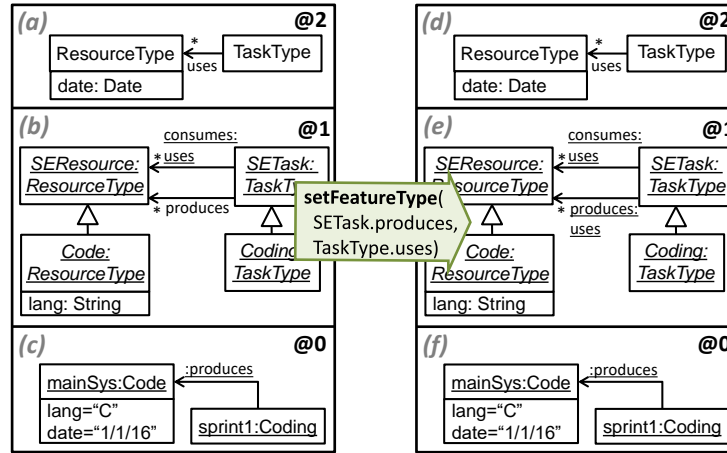


Fig. 27: Example of refactoring *set feature type*: produces gets typed by uses

ontological type of C or a superclabject of it, should have the potency of f plus 1, and its maximum cardinality should be either unbounded or bigger than the number of its instances. If f and f' are fields, their data types should be the same. If they are references, the target of f should be an instance of the target of f' or a subclabject.

Working scheme. Refactoring *setFeatureType*(C.f, C'.f'), where f is the feature to type by feature f', works as follows:

(1) Set f' as the type of f.

While the refactoring algorithm is trivial, the pre-condition checking is complex.

A.10. Split clabject

This refactoring splits a clabject into two: one with the features with potency 1, and the other with the features with potency bigger than 1. The latter clabject is moved one meta-level down, and becomes an instance of the former.

Type. Top-down; might be level expanding; atomic; no semantic side effects.

Utility. A clabject has been attributed features that are not useful in all domains or expected instances, though some of its features are truly general. To fix this issue, the refactoring moves the domain-specific features (i.e., with potency bigger than 1) to a new clabject in the lower meta-level. The repeated application of this refactoring ultimately removes potencies bigger than 1 from a multi-level model, and can be used to migrate to systems that do not support deep characterization.

Consequences. The split clabject does no longer deeply characterize instances beyond the meta-level below. Its features with potency bigger than 1 are pushed down, and appropriate clabject representatives are created (see Section 5.2).

Example. In Figure 28, ResourceType in model (a) defines some features which are not applicable to every process modelling domain. Applying the refactoring to the clabject creates the new clabject SEResource as an instance of it, and as superclabject of the existing instances of ResourceType (Reqs and ProgDoc). The features of ResourceType with potency bigger than 1 (lang, date and usedBy) are pushed down into SEResource with decreased potency. Reference ResourceType.usedBy can be pushed down to SEResource because the reference has no instances. Moreover, an abstract clabject SETask is created as target of the pushed-down reference to preserve the semantics of the multi-level model.

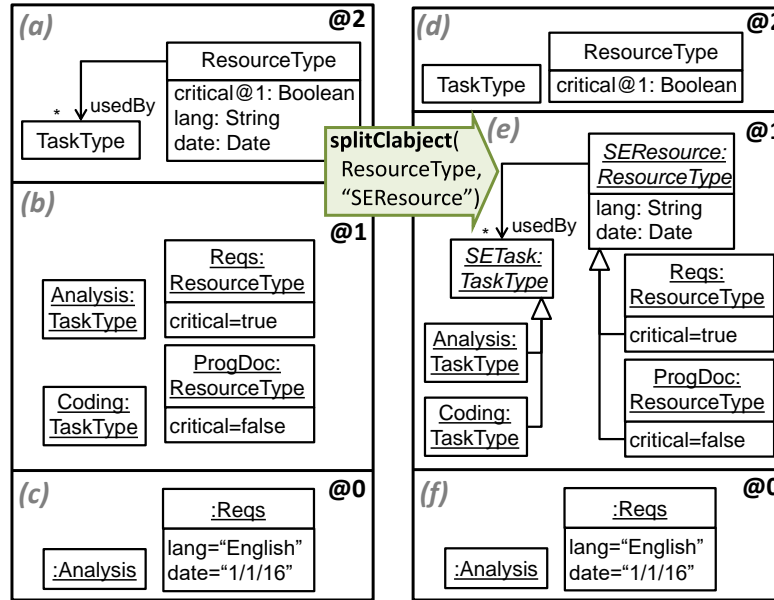


Fig. 28: Example of refactoring *split clabject*: ResourceType at level 2 is split and results in the creation of SEResource at level 1

Pre-conditions. The clabject to be split cannot be abstract or have potency 0. The references with potency bigger than 1 owned by the clabject cannot have instances.

Working scheme. Refactoring *splitClabject*(C, name), where C is the clabject to split, and name is the name of the introduced clabject, works as follows:

- (1) Let M be the container model of C, and M' an instance of M.
- (2) If M' does not exist, create it.
- (3) Create a clabject C' in M' with the potency of C minus 1, and named name.
- (4) Set the type of C' to C.
- (5) If C has instances, make C' abstract.
- (6) Set C' as the superclabject of each instance of C or any subclass of it.
- (7) For each feature f in C with potency bigger than 1:
 - (a) apply refactoring *pushDownFeature*(C.f, {C'}).

In the first step, the refactoring identifies the model M' where the new clabject will be created. In practice, this is obtained from the context model where the refactoring command is issued. If no such a model exists, it is created in step 2, hence causing a level expansion. Then, a clabject C', instance of C, is created (steps 3 and 4). C' becomes the representative of all existing instances of C and its subclass of it (step 6). Finally, the features of C with potency bigger than 1 are pushed down to C' using refactoring *push down feature* (step 7). This refactoring may create clabject representatives of the instances of the target of the pushed-down references (see Section 4.3), but it does not produce side effects because C' is the representative of all instances of C and its subclass of it.

A.11. Stratify potency

This refactoring reorganizes the features of a clabject according to their potency along a hierarchy of abstract clabjects with decreasing potency. Note that, as stated in [de Lara et al. 2014], a clabject can only inherit from a clabject with less or equal potency.

Type. Intra-level; level preserving; atomic; no semantic side effects.

Utility. When a clobject owns features with different potencies, organising the features in a hierarchy of clobjects according to their potency permits a more flexible reuse of the features via inheritance.

Consequences. The refactored clobject C inherits from a hierarchy of abstract clobjects with different potencies. The top-most clobject in the hierarchy has potency 1 and contains all features of C with potency 1; its child has potency 2 and contains all features of C with potency 2; and so on until the potency of C is reached. The hierarchy of clobjects provides flexibility as, e.g., other clobjects with potency 1 can be set to inherit from the top-most clobject in the hierarchy.

Example. In Figure 29, the designer requires modelling different types of resources in addition to SoftwareType resources. To have flexibility when defining the new resource types, the designer applies this refactoring to SoftwareType. The refactoring creates the abstract superclobject ResourceType with potency 1 and containing all features of SoftwareType with potency 1 (i.e., critical). In this way, the designer can define other types of resource by adding new subclobjects of ResourceType or SoftwareType as required according to their features.

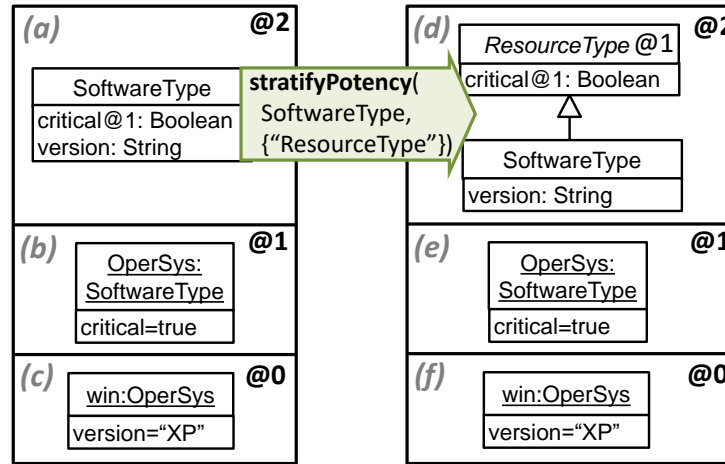


Fig. 29: Example of refactoring *stratify potency*: SoftwareType is divided into two clobjects

Pre-conditions. The clobject to stratify cannot have superclobjects or potency 0.

Working scheme. Refactoring *stratifyPotency*(C, {name₁, ...}), where C is the clobject to stratify, and {name₁, ...} is the collection of clobject names in the resulting hierarchy, works as follows:

- (1) Let current = C.
- (2) For i=C.potency-1 down to 1:
 - (a) create an abstract clobject D_i with name name_i and potency i.
 - (b) move to D_i all features of C with potency i.
 - (c) if C is the target of a moved reference, set the target to D_i.
 - (d) set D_i as the superclobject of current.
 - (e) set current = D_i.

The algorithm iterates each potency value from the potency of C minus 1 down to 1, creating a clobject for each potency value that contains C's features with that potency. The created clobjects are arranged in an inheritance hierarchy with clobject D₁ with

potency 1 on top, and C on bottom. If a reference in C gets moved to some clabject D_i , then its target is also set to D_i .

A.12. Unset clabject type

This refactoring unsets the ontological type of a clabject, making it a linguistic extension. It has two variants. In the main one, the clabject does not retain the fields declared by its type with potency bigger than 1. In the alternative variant, the clabject keeps all features.

Type. Top-down; level preserving; atomic; one of its variants has semantic side effects.

Utility. The ontological type of a clabject is not a good classifier for it. Hence, applying this refactoring removes the instantiation relation between them.

Consequences. The clabject drops its ontological type and becomes a linguistic extension.

In the main variant, the instances of the clabject may need to be repaired to delete all instances of fields declared in the clabject's ontological type with potency bigger than 1; hence, this variant may have side effects. In the alternative variant, the clabject and its instances retain all features; therefore, this variant has no semantic side effects.

Example. In Figure 30, SwEngineer in model (b) is an instance of ResourceType, and jdoe in model (c) is an instance of SwEngineer and can assign a value to field date as this is defined by ResourceType with potency 2. However, the designer plans to create a new clabject Actor and type SwEngineer by this new clabject, which is a better classifier. Hence, as a first step, the designer applies the refactoring to SwEngineer to unset its type. SwEngineer becomes a linguistic extension. Since the designer applies the main variant of the refactoring, the fields with potency bigger than 1 in ResourceType become unavailable to the instances of SwEngineer. This is the case for field date, so the refactoring removes its slot from clabject jdoe (see model (f)). The ontological type of reference SwEngineer.requiredFor is unset, but this has no effect in the instances of SwEngineer.

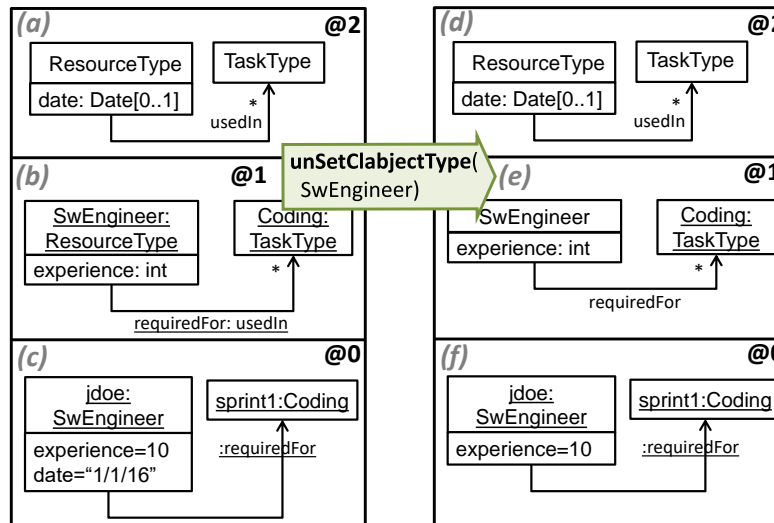


Fig. 30: Example of refactoring *unset clabject type*: clabject SwEngineer becomes a linguistic extension

Pre-conditions. The clobject to untype should have a type, and cannot have superclobjects or subclobjects, or be the target of references having a type.

Side effects. The main variant has side effects when the ontological type of the clobject has fields with potency bigger than 1, as the example of Figure 30 shows.

Working scheme. Refactoring `unSetClObjectType(C)`, where C is the clobject to untype, works as follows:

- (1) Let C' be the ontological type of C.
- (2) Unset the type of C.
- (3) Let F be the set of features owned or inherited by C' with potency bigger than 1.
- (4) For each feature f in F:
 - (a) apply refactoring `unSetFeatureType(C.f)`.
 - (b) **[main]** if f is a field (with primitive type):
 - i. **[main]** remove f from C.
 - ii. **[main]** for each direct and indirect instance l of C, remove f from l.

The algorithm unsets the type of C in step 2. Then, it unsets the type of the features of C that are instances of features with potency bigger than 1 in C's ontological type (step 4a). This is done by applying refactoring *unset feature type* (Section A.13), which does not have side effects because the clobject has no type at this point. In addition, the main variant deletes the identified features and its instances from C and its instances, but only in case of features with a primitive type (step 4b).

A.13. Unset feature type

This refactoring unsets the ontological type of a feature.

Type. Top-down; level preserving; atomic; it may have semantic side effects.

Utility. The ontological type of a feature is not a good classifier for it. A feature (typically a reference) was initially thought to be an instance of another, but it is not. Applying this refactoring removes the instantiation relation between both features.

Consequences. The feature drops its ontological type and becomes a linguistic extension, but it retains its potency and cardinality. The result of the refactoring may violate the minimum cardinality of the ontological type of the feature, as the pre-conditions do not check this. In case of fields, unsetting the type has side effects because the fields need to be renamed to avoid name collisions of fields in the instance and type clobjects.

Example. In Figure 31, the designer decides to unset the type of reference `SwEngineer.requiredFor`, which becomes a linguistic extension. The existing instance of `SwEngineer` is not affected by this change.

Pre-conditions. The feature to untype should have a type. If it is a field, its new name should not collide with existing field names in the clobject or its direct and indirect instances.

Side effects. This refactoring has side effects when the feature is a field as, in this case, the field needs to be renamed to avoid naming collisions of fields in the instance and type clobjects.

Working scheme. Refactoring `unSetFeatureType(C.f, name?)`, where f is the feature to untype and name its (optional) new name, works as follows:

- (1) Unset the type of C.f.
- (2) If C.f is a field (with primitive type):
 - (a) set the name of C.f to name.
 - (b) set the name of direct and indirect instances of C.f to name.

The refactoring unsets the type of the feature, and in case of fields, they are renamed. The latter causes a side effect.

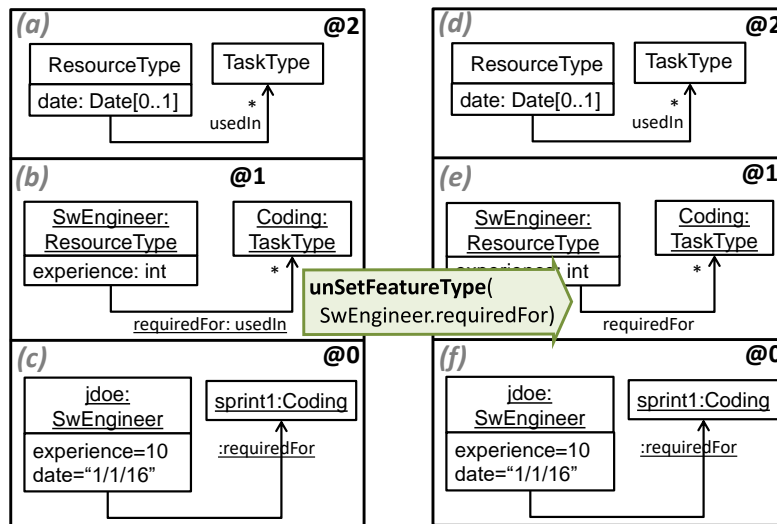


Fig. 31: Example of refactoring *unset feature type*: reference **requiredFor** becomes a linguistic extension