Challenges in the Evolution of Metamodels: Smells and Anti-Patterns of a Historically-Grown Metamodel

Misha Strittmatter
Chair for Software Design and Quality (SDQ)
Karlsruhe Institute of Technology
Karlsruhe, Germany
strittmatter@kit.edu

Georg Hinkel
Software Engineering Division
FZI Research Center of Information Technologies
Karlsruhe, Germany
hinkel@fzi.de

Michael Langhammer
Chair for Software Design and Quality (SDQ)
Karlsruhe Institute of Technology
Karlsruhe, Germany
langhammer@kit.edu

Reiner Jung
Software Engineering Group
Christian-Albrechts-University Kiel
Kiel, Germany
reiner.jung@email.uni-kiel.de

Robert Heinrich
Chair for Software Design and Quality (SDQ)
Karlsruhe Institute of Technology
Karlsruhe, Germany
heinrich@kit.edu

ABSTRACT

In model-driven engineering, modeling languages are developed to serve as basis for system design, simulation and code generation. Like any software artifact, modeling languages evolve over time. If, however, the metamodel that defines the language is badly designed, the effort needed for its maintenance is unnecessarily increased. In this paper, we present bad smells and anti-patterns that we discovered in a thorough metamodel review of the Palladio Component Model (PCM). The PCM is a good representative for big and old metamodels that have grown over time. Thus, these results are meaningful, as they reflect the types of smells that accumulate in such metamodels over time. Related work deals mainly with automatically detectable bad smells, anti-patterns and defects. However, there are smells and anti-patterns, which cannot be detected automatically. They should not be neglected. Thus, in this paper, we focus on both: automatically and non-automatically detectable smells.

Keywords

metamodel smells; metamodel maintainability; metamodel anti-patterns; Palladio Component Model

1. INTRODUCTION

Model-driven engineering uses domain-specific modeling languages (DSMLs) to express abstractions of reappearing domain concepts. DSMLs can then be used to design systems. The resulting models (instances of DSMLs) can be analyzed and used for simulation. Depending on the DSML, its instances can be used for purposes ranging from stub generation to the generation of fully functional code. DSMLs are defined by metamodels.

The evolution of metamodels is a big challenge, as they tend to be central artifacts with many tools that depend on them. A change in a metamodel may cause many errors in dependent code. Thus, it is important that metamodels allow the addition of new features by creating metamodel extensions (which we will refer to as *external extensibility*). Some bad smells in metamodels make external extensions impossible. Other bad smells degrade a metamodels general maintainability.

In this paper, we present the results of a thorough metamodel review of the PCM in the form of a list of bad smells. The PCM [26] is a metamodel to describe component-based software architectures with a focus on their performance properties. It is implemented in EMF's Ecore [29], which is an implementation of EMOF [24]. The metamodel is approximately 10 years old and has been extended by features over time. While later features have been extended externally, earlier additions were made intrusively. Thus, the PCM is a representative specimen of a big, old and grown over time metamodel. That is the reason why we think the results of our review are important to the community. They show what type of smells typically accumulate in metamodels over time.

Related work mainly focuses on the automatic detection of metamodel smells and simple defects of the metamodel. Being able to detect these problems automatically is important. However, smells and anti-patterns that cannot be automatically detected should not be ignored. Thus, besides four bad smells that can be automatically detected, we also present 6 smells which can only be detected manually. This paper does not cover any metamodel defects, which break conformance to the meta-metamodel (comparable to compiler errors in programming). These defects are usually taken care of by the modeling frameworks validation functionality.

This paper is structured as follows: Section 2 gives foundational information about metamodels, their evolution and maintainability, anti-patterns and bad smells. Section 3 presents the state of the art in evaluating metamodel quality and detecting bad smells in metamodels. Section 4 will briefly give background information and present the internal structure of the PCM. Section 4 will show how the PCM has developed over the year. Section 5 contains the list of bad smells in the PCM. Section 6 concludes the paper.

2. FOUNDATIONS

In this paper, we are concerned with Essential Meta-Object Facility (EMOF)-conforming metamodels [24] (e.g. instances of the EMF's Ecore meta-metamodel). A *metamodel* defines and constrains the set of its instances (i.e. models). In the sense of EMOF, a metamodel consists of metaclasses, which in turn contain relations and attributes. If the elements of a model conform to the definitions in the metamodel, the model is an instance of the metamodel. EMOF-conforming metamodels are similar to UML class diagrams. The differences are that they have to be complete and have to form a containment tree.

The relations that can be defined in a metamodel connect two metaclasses. Attributes of metaclasses have only primitive types. Metaclasses are able to inherit from each other. A special case of a relation is the containment relation. Each element of a model, except its roots, has to be contained in another element. A metamodel can be subdivided into packages. A metamodel may also reference metaclasses of another metamodels. Constraints can be defined (e.g. using the Object Constraint Language (OCL) [25]).

Metamodels evolve because their languages evolve. New features have to be added, concepts have to be adapted, and bugs have to be fixed. A more detailed classification of metamodel evolution can be found in [31]. In our experience with the PCM, the biggest driver is the inclusion of new features.

How easy it is to evolve a metamodel can be considered the *maintainability* of a metamodel. The maintainability is influenced directly by a metamodel's complexity and understandability. Understandability is not completely derived from complexity, as it is possible to metamodel a simple concept in a way which is not intuitive. The concepts of cohesion [36] and coupling [4] can be transferred from object-oriented software development. The *cohesion* of a module is described as how related its classes are. *Coupling* of one module to another expresses how dependent the first module is on the second. Both measures are heuristics for maintainability. A high cohesion is beneficial, as related classes tend to evolve together. A high coupling between packages is detrimental, because modifications may have a bigger impact on dependents.

In object-oriented software development, a *bad smell* [9] is considered an indicator for a possible problem in the software's design or code. An anti-pattern was originally defined as being "... just like pattern, except that instead of a solution it gives something that looks superficially like a solution, but is none." [27]. However, the meaning of *anti-patterns* changed over time to mean a recurring pattern that has negative consequences [28, 14], regardless if it was purposely used or not. When transferring these terms to the domain of metamodeling, some bad smells can be defined as anti-patterns [2]. Other bad smells may be indicated by metrics [2]. Some are only detectable by manual investigation. In the following, we will refer to bad smells in metamodels as *metamodel smells* or simply *smells*. Metamodel smells may have various negative effects on metamodel maintainability, which we will explain in this paper. In our list of smells, we include automatically detectable smells, but even more importantly smells which can only be reliably detected manually. As the PCM is a valid metamodel, metamodeling errors (which prevent the generation of the model code) will not appear in our list. For each smell we identified, we explain the characteristics of this smell, its consequences, reasons why they appeared in the PCM, how we think they can be best corrected, whether we can automatically detect them and where they occurred in PCM.

3. STATE OF THE ART

EMF Refactor [2, 1] is a tool that can be used to automatically detect bad smells and perform refactorings in Ecore-based metamodels and UML models. They detect bad smells either by a violation of a specific metric or by the detection of an anti-pattern. For Ecore they feature automated detection of the following anti-patterns[1]: Large EClass, Speculative Generality EClass, Unnamed EClass. They feature an even longer list for UML anti-patterns. Of these, some may also be applicable to Ecore metamodels.

Elaasar [7, 8] developed an approach for automated detection of patterns and anti-pattern in MOF-based models. His approach provides a ready to use catalog with patterns specifications but also supports the creation of new pattern specifications by the user. His MOF anti-patterns are grouped in the categories well-formedness, semantic and convention.

López et al. propose a tool and language to check for properties of metamodels [20]. In their paper, they also provide a catalog of properties, which they categorize in: design flaws, best practices, naming conventions and metrics. They check for breaches of fixed thresholds for the following metrics: number of attributes per class, degree of fan-in and -out, depth of inheritance tree and the number of direct subclasses.

Vépa et al. present a repository for metamodels, models, and transformations [35]. The authors apply metrics that were originally designed for class diagrams onto metamodels from the repository. For some of the metrics, Vépa et al. provide a rationale how they relate to metamodel quality.

Di Rocco et al. [6] applied metrics onto a large set of metamodels. Besides the usual size metrics, they also feature the number of isolated metaclasses and the number of concrete immediately featureless metaclasses. Further, they searched for correlations of the metrics among each other. E.g., they found that the number of metaclasses with superclass is positively correlated with number of metaclasses without features. Based on the characteristics they draw conclusions about general characteristics of metamodels. Their long-term goal is to draw conclusions from metamodel characteristics concerning the impact onto tools and transformations that are based on the metamodel.

Gómez et al. [13] propose an approach, which aims at evaluating the correctness and expressiveness of a metamodel. I.e. weather it allows invalid instances (correctness) and is it able to express all instances it is supposed to (expressiveness). Their approach automatically generates a (preferably small) set of instances to evaluate these two criteria.

García et al. [11] developed a set of domain specific metamodel quality metrics for multi-agent systems modeling languages. They propose three metrics: availability, specificity and expressiveness. These metrics take domain knowledge into account, e.g., the "number of necessary concepts" or the "number of model elements necessary for modeling the

[1]https://www.eclipse.org/emf-refactor/index.php

system of the problem domain".

There is much work on quality metrics for object-oriented design and UML class diagrams [5, 22, 21, 12]. Further, there are publications that present empirical analyses of object-oriented design metrics [3, 34]. E.g. Subramanyam found that the correlation between metrics and bug detection varied when applied to different programming languages and observed interactions between metrics. The purpose and usage of object-oriented design and class diagrams is very different compared to metamodels, thus their benefit cannot be assumed for metamodels.

4. THE PALLADIO COMPONENT MODEL

The Palladio approach [26] is an approach to component-based software engineering. At its core is the PCM, a metamodel, which defines a language to express component-based software architectures and abstractions of several quality aspects. In this section, we present insights into the structure of the PCM.

The PCM is separated in different hierarchical packages. The root package is called `pcm` and directly or indirectly contains the remaining packages. In this paper, we consider packages, that are directly contained in the root packages as first level packages. Packages contained in first level packages are considered as second level packages and so on. The containment hierarchy of the packages is depicted in Figure 1, while dependencies between the packages implied by inheritance of classes within the packages are depicted in Figure 5.

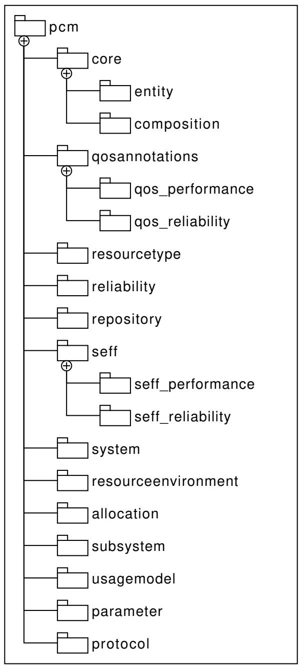

Figure 1: The package containment hierarchy of the PCM

In [32], we identified different main concerns of the PCM, which can be seen in Figure 2. The figure shows dependencies how they should be, not how they actually are. As one can see, in these two figures, the packages are mainly sliced as the concerns. For instance, the `repository` package contains all classes that are necessary to build a `Repository` with its `DataTypes` and `ComponentTypeHierarchy`. However, some packages contain information for multiple concerns.

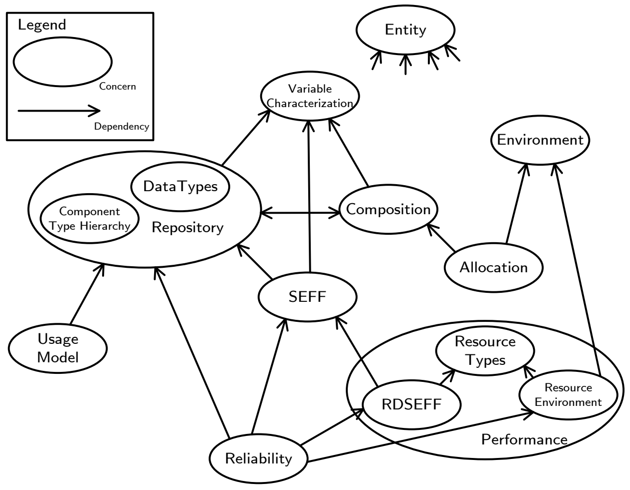

Figure 2: The main concerns of the PCM [32]

Starting in August 2006, the PCM has a long evolution history. Since then, many new features have been added. Although Palladio initially was built for performance prediction, it now also supports to simulate reliability, data consistency, energy consumption and maintainability. At the same time, besides the original call characteristics, a system can now be specified in an event-based manner and PCM has built-in support for extensions in the form of stereotypes.

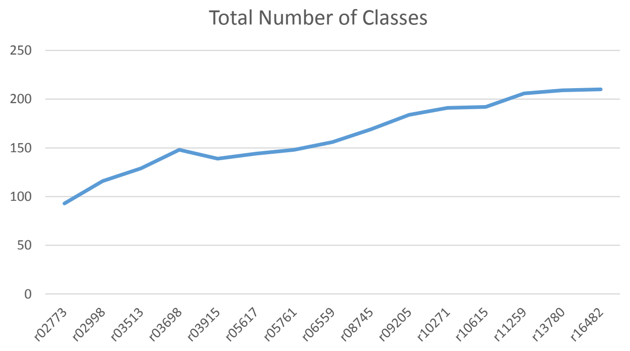

Figure 3: Evolution of the Palladio Component Model between March 2007 and September 2012 in terms of the number of classes

Many of these perfective changes have had a footprint to the metamodel. Today, the version control system registers more than 120 revisions of the core metamodel plus several revisions for underlying shared functionality it is using. To give an impression on the evolution of the metamodel, we have depicted the total number of classes between spring 2007 and fall 2012 in Figure 3. Most changes to the PCM were mad in this period and the metamodel size in terms of number of classes has more than doubled.

5. METAMODEL SMELLS

In this Section, we will present the list of metamodel smells we found in the most recent version of the PCM. This version of the metamodel was released with the Palladio Bench 4.0. For each metamodel smell, the following aspects will be elaborated: general description, its consequences, reasons for its forming, or even rationale why it might have been purposefully used, possible resolutions, if the smell is automatically detectable as well as brief mention of its occurrences in the PCM. In a smell's description, we will also briefly discuss the relation of a smell to object-oriented design and programming (OO).

5.1 Redundant Container Relation

Description: Containment relations are necessary to be able to specify entities that are more complex and they guide serialization. To navigate models, it can be necessary to traverse from a contained element to the containing element. For this purpose, the container reference can be used. In EMF, this feature is provided by a generic and implicit reference eContainer. However, it is also possible to define an explicit container reference utilizing the concept of opposite references.

This smell does not exist in OO, as there are in general no explicit containers. Objects are contained in the heap memory and are merely referenced by other objects. If no more references to an object exist, the object is eventually deleted. In metamodeling and modeling on the other hand, an EObject must have a container. If that container is deleted, all contained elements are also deleted and all references to the deleted elements are unset.

Consequence: This smell has several negative implications. First, it introduces redundancy, as the implicit eContainer reference is always present and an explicit container reference is a duplication. Second, it increases metamodel complexity due to this duplication. In case a class is used in different containments, multiple explicit container references exist. These references are all mutually exclusive, but this cannot be declared with EMF itself. Furthermore, the opposite references must be declared as optional, which weakens their meaning. For example, the `PCMRandomVariable` of the PCM metamodel is used in 17 contexts. Therefore, the class has 17 opposite references, but only one is used by an instance. Third, the explicit container reference can harm reuse and evolution of metamodels. In case a container class is added or removed, this always also requires the adaption of the contained class. In case different aspects and partitions of metamodels are stored separately, a cyclic reference between the metamodel of the containing class and the contained class is necessary. This hinders reuse, as both metamodels must be present. However, using an implicit eContainer reference, the metamodel with the contained class can be reused in other contexts.

Reason and Rationale: Some may argue that container references allow ensuring static type safety. However, this only applies for cases with only one container reference and realized in Java directly. In case of multiple containing classes, the static type safety property is weakened, as the containing class type cannot be determined statically. Instead, at runtime each property must be checked which is similar to testing the type of the containing class, but it is obfuscated that this is indeed a type check.

In some UML metamodels, associations are used where both ends are named, and one end is declared as composite. If this property of the UML metamodel must be preserved, an explicit container reference cannot be omitted. However, such naming can exist only for documentary reasons, which allows ignoring them for the EMF mapping of the UML metamodel.

Correction: The explicit container reference can be removed and its usage in code can be replaced by accessing the eContainer reference. If only the explicit container reference was used before, the eContainer reference can be safely cast to that container. If, however, the explicit container reference is checked for null, the eContainer reference has to be checked for the type of the expected container.

Automatic Detection: Opposite references for containment references can be detected automatically.

Occurrences: In the PCM, 85 explicit container references can be found with 17 of these references originating from the aforementioned `PCMRandomVariable`.

5.2 Obligatory Container Relation

Description: This smell is a special case of the redundant container relation smell (Section 5.1). If a containment relation has an opposite reference that has a lower bound of 1, we call it an obligatory container relation. This is illustrated in Figure 4. Class `C1` contains `A` and the container relation is obligatory. As with the redundant container relation smell, this smell is not relevant in OO, as there is no explicit containment.

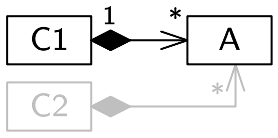

Figure 4: Obligatory Container Relation for Classes C1 and A

Consequence: `A` cannot be used in any other context. E.g., although `C2` has a containment relation to `A`, an instance of `C2` can never contain any instances of `A`, as the container relation to an instance of `C1` has to be set. In such cases, the EMF framework does not even allow code generation.

Reason and Rationale: There are some reasons to use an obligatory container relation. It ensures type safety when navigating to the container. It is also possible, that the developers want to restrict reuse explicitly. However, in most circumstances the developers were most likely unaware of these consequences. Obligatory container relations can also be the result of translation from another format or language (e.g., UML) by a transformation.

Correction: To fix this smell, remove the container relation (this will also resolve the redundant container smell). As the class could only be in one type of container, the eContainer can be safely cast to that container class.

Automatic Detection: Obligatory container relations can easily be detected. However, manual evaluation is still required, as in some cases they may be intended by the developer.

Occurrences: In the PCM, many container relations are obligatory. We suspect this to be the result of the transformation from Rational Software Architect.

5.3 Concern Scattered in Package Hierarchy

Description: The package hierarchy of a metamodel is mainly for logical partitioning of its content. We consider it a bad smell, if the classes that constitute a feature or concern of the language are spread over multiple packages. Even crosscutting concerns can be modularized in a more meaningful way.

In OO, there are issues similar to this smell, e.g., when cohesive classes are scattered over packages or assemblies. However, to our knowledge, there is no explicit smell that covers the problem on this level. OO smells are more concerned with the internals of packages: relations between classes and the internals of classes and methods.

Consequence: This bad smell has negative consequences on the understandability and thus maintainability of metamodels. When a developer tries to understand a metamodel, he examines its packages and from their content and documentation (if there is any) tries to conclude its purpose. If a concern is scattered, the purpose of the package cannot be fully comprehended without tracing relations that leave the package. The smell may also increase coupling between affected packages and reduce relative cohesion within the packages.

Reason and Rationale: We suspect that this smell occurs mostly, when new concerns are implemented in an already existent metamodel. The new concern is related to the concerns of multiple other packages. Parts of the new concern are then placed in the packages of the related concerns and so the new concern is ripped apart.

Correction: A better approach would be to place the new concern in its own package. The package should further contain sub-packages for each related concern, which then contain the classes that are related to these concerns. The package of the new concern should be placed meaningfully. If it is a first order concern, it should be placed below the root package. If it is a subconcern, its package should be placed as a subpackage of the parent concern. If it is a crosscutting concern, it should be placed on the same level, as the concerns it is intersecting with.

Moving classes can be done through refactorings. Even the code, which depends on the classes, may be automatically fixed. A mere moving of affected classes may lead to other bad smells, if the dependencies are not modified. The new dependencies between packages may lead to package dependency cycles (see Section 5.5) and violations of the dependency inversion principle on the package level (see Section 5.6). This is not the fault of consolidating a concern, but of dependencies that were improper in the first place. An explicit reference structure (e.g., [33, 30]) can help in structuring packages and directing their properties properly.

Automatic Detection: This bad smell is not automatically detectable. An algorithm is not able to automatically infer the semantics of parts of the metamodel.

Occurrences: It is difficult to nail down the exact number of occurrences in the PCM, as this depends on how fine-grained its concerns are identified. Looking at quite coarse-grained concerns, there are at least six occurrences of this smell in the PCM [32]. The following concerns are affected: resource interfaces, middleware infrastructure, performance, repository (especially interfaces), event communication and reliability.

5.4 Multiple Concerns in Package

Description: Conversely to the scattered concern smell, we also consider it a smell, if a package contains the classes of multiple concerns. The relation of this smell to OO is analog to the scattered concern smell. Insufficient modularization on the package level is an issue in OO. However, we are not aware of an explicit smell definition.

Consequence: Having multiple concerns in one package, increases the effort to understand the package, because the developer has to identify the contained concerns and their respective classes. Simply put, the package is needlessly complex. This bad smell might also decrease the cohesion within the package.

Reason and Rationale: We suspect that this smell has two explanations. Developers tend to place classes in packages, which hold their container or represent a closely related concern. It is just more convenient to use the existing package hierarchy than to think of a new structure yourself.

Correction: How to modularize and package concerns is already well explained in the resolution part for the scattered concern smell (see Section 5.3). As already suggested, new concerns should be placed in their own package. If a concern is a subconcern, then its package should be placed as a subpackage.

Automatic Detection: This bad smell is not automatically detectable. An algorithm is not able to automatically infer the semantics of parts of the metamodel.

Occurrences: There are at least two occurrences of this smell in the PCM [32]. The following concerns are affected: data types and the abstract component type hierarchy, which both are located in the repository package.

5.5 Package Dependency Cycles

Description: When creating a metaclass, one of the most important choices is the selection of appropriate base classes, from which some functionality can be reused. However, inheritance is a white-box technique. Therefore, one needs to fully understand the base classes. Hence, a closer look into the containing packages is required. This dependency between packages implied by inheritance can be shown visualized as a graph. We depicted this graph for the latest PCM version in Figure 5. This graph may contain cycles (shown in red).

In OO, having dependency cycles in assemblies is consid-

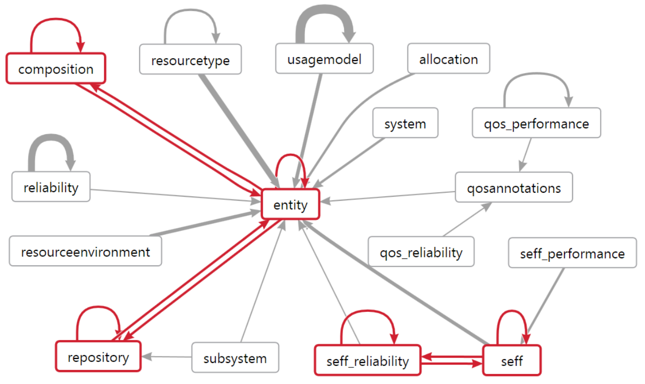

Figure 5: Dependencies between packages in PCM implied by inheritance of classes, thickness of arrows indicates how many classes use an inheritance relation, cycles are marked in red.

ered a bad practice, or is even treated as an error on some platforms. However, there is not so much emphasis placed on dependency cycles on the package level.

Consequence: A consequence of such a circular dependency may be that to fully understand a package contained in such a circle, a developer has to understand all packages contained in this circle. This challenges the appropriateness of the package structure.

Especially if the cycle is formed from inheritance dependencies, the maintainability of the metamodel may suffer. Changes made to the metamodel propagate down the class hierarchy and thus into other packages where they should not.

Reason and Rationale: PCM makes extensive use of multiple inheritance and the inheritance hierarchy of some metaclasses is quite high. Therefore, it may have become difficult to keep an eye on package dependencies.

Correction: In situations when developers have lost an overview of the package dependencies, we think that an overview such as in Figure 5 can already be helpful to avoid this anti-pattern.

Automatic Detection: A circular dependency can be detected automatically (in fact, Figure 5 is entirely generated by a tool) and could be even automatically resolved by merging the affected packages. However, we think a manual inspection can be more beneficial in such a scenario.

Occurrences: The occurrences of this pattern in the latest version of Palladio is shown in Figure 5.

5.6 Dependency Inversion Principle Violated

Description: The dependency inversion principle [23] is a design principle from OO. When translated to metamodeling it states that high-level classes should not depend on low-level classes (high- and low-level regarding the level of abstraction). Both may depend on abstractions. The same can be said about packages and even metamodels, when the dependencies of a package or metamodel are regarded as the combined dependencies of their elements.

Part a) of Figure 6 shows a violation of the principle on the class level. Package H contains high-level concepts, relative to which the content from the package L is low-level. Class A has a dependency (relation, inheritance or containment) to K. Thus, a high-level class is dependent on a low-level one. Class K contains further information about A (indicated by the data attribute). Although it is illustrated as a single attribute, this information may come in the form of attributes, relations and containments.

Consequence: A violation of the dependency inversion principle may have detrimental effects on the maintainability of a metamodel. During evolution, modifications of a concern may influence a more high-level concern. Such violations do also hinder understanding. When a developer tries to understand a concern, he may trace the outgoing relations to more low-level concerns. Thus, he may examine concerns which are not necessary for understanding the high-level concern or even irrelevant to his intent.

Reason and Rationale: In our opinion, these violations stem from the integration of features. It is most convenient for a developer to extend an existing class hierarchy by adding dependencies that point to the new content. However, there is a difference between object-oriented design and metamodels. In OO, it is easier and more natural to introduce new abstraction layers. In metamodels, on the other hand, interfaces cannot be used in a similar way, because usually it is not similar functionality that is added, but new and different data. At first glance, it seems to be a good solution just to create a new subclass, which adds the needed information. However, when multiple new features are implemented this way, they cannot be combined. Therefore, in metamodeling, it can be reasonable to violate the dependency inversion principle in certain cases. A possible rule would be to do so for core features of the language. A feature can be considered a core feature, if it is useful in every use case of the language and for every possible type of user. Core features should be integrated intrusively into the metamodel with a violation of dependency inversion principle. This has the following advantages: type safety, adherence to cardinality and retrieval in $O(1)$ (constant time).

Correction: When implementing non-core features, the dependency inversion principle should be used. In Figure 6, we point out two possible options.

In b), the new abstract class B is created as well as a con-

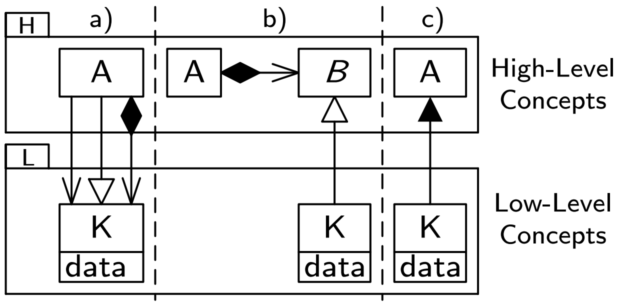

Figure 6: Violation and Application of the Dependency Inversion Principle

tainment from `A` to `B`. Class `K` then inherits from `B`. Thus, the dependency is reversed and now goes from `L` to `H`. This solution has some benefits. The instances of `K` are contained in instances of `A`. This enables direct navigation and thus retrieval in constant time. In addition, the cardinality can be controlled directly without having to specify complex constraints. However, type safety is not guaranteed, as the extended data is not placed in `B` but in `K`. This solution has to be enabled in the initial development, as the class `B` is required. This is no issue, if `K` is also already created during the initial development or if the future extension of `K` can be foreseen. The main disadvantage of this solution is that `H` has to be modified, if this solution should be implemented in hindsight.

In `c`), an alternate solution is shown which does not require modification of `H`. By either stereotype application [18, 17], aspect-oriented extension [15, 16] or plain referencing, instances of `K` can be associated with instances of `A`. Compared to `b`) this has the disadvantage, that the look-up is in $O(k)$, where k is the number of instances of `K`.

Automatic Detection: This smell is not automatically detectable. An algorithm is not able to deduce if concepts are higher- or lower-leveled compared to others.

Occurrences: Within the PCM is at least one serious occurrence of this smell with regard to the inheritance hierarchy. The superclass that represents entities that require and provide interfaces, inherits from a superclass which represents entities which require and provide resource interfaces. While the usage of interfaces is a core feature of the PCM, resource interfaces are not. Regarding violations with references, there are countless occurrences. They stem from the intrusive integration of features into more abstract concepts.

5.7 Dead Class

Description: As a result of a refactoring, in some cases a class is no longer required as its responsibilities are taken care of by another class. Sometimes, although the references to the class are deleted, the class itself is not. In OO, this smell falls under the category of dead code or oxbow code.

Consequence: This has a negative impact on understandability since the class has to be considered, even though it cannot be contained in the rest of the model. Furthermore, developers may have a hard time trying to understand how the class is used. When they finally find out that it is not used at all, this has an impact on their opinion of the metamodel.

Reason and Rationale: The pattern is mainly a result of bad metamodel reuse. As the metamodel gets large, it is no longer obvious in which places a class is used. Thus, when the class is no longer required in one specific scenario, it still may be required in another. However, as developers do not check whether the class is used in some other place, the class may be left behind with no usage elsewhere in the metamodel.

Correction: This problem can be avoided if developers make sure that classes they no longer need are either still used elsewhere or deleted.

Automatic Detection: It is possible to statically detect that a class cannot be contained in another class. However, a manual assessment is then required to decide whether this class is dead, as for root container classes, it is viable (though not obligatory) to be uncontainable.

Occurrences: A static analysis of PCM delivers in total nine uncontainable classes. From these, the classes **`UsageModel`**, **`Repository`**, **`ResourceRepository`**, **`System`**, **`ResourceEnvironment`** and **`Allocation`** represent view types and therefore serve as model roots. Thus, it is perfectly valid for them to be uncontainable. The class **`DummyClass`** has been introduced to overcome a technical limitation in the QVT-O compiler, but this is a rather different issue (the purpose of this class is hardly documented, but developers trying to understand PCM would not expect any reasonable semantics from a class named like this). However, over the history, two classes have been left over from refactoring operations, **`CharacterisedVariable`** and **`ResourceInterfaceProvidingRequiringEntity`**. For both of these cases, it is not obvious that they are no longer needed, so developers may try to find usages and fail to do so.

5.8 Concrete Abstract Class

Description: This smell is concerned with classes that should be abstract, but are not. Usually, in a class hierarchy, a class with subtypes is abstract. However, not every occurrence is necessarily bad design, as sometimes even a concrete class might have concrete subclasses.

In OO, having a concrete abstract class is also a problem. However, we are not aware of an existing smell definition.

Consequence: However, if a class that should be abstract is not declared as such, this has a negative impact on the metamodels correctness and understandability. Due to the fact, that an instance of the metamodel may validly contain direct instances of a class that should not have any instances, the metamodel is less correct. Usually this problem is hidden by self-built model editors, which just do not offer any possibility to create direct instances of the affected class. However, using fully generated model editors (like the EMF tree editors), this problem does manifest. Further, the understandability of the metamodel is slightly reduced by this smell. A developer, who investigates the metamodel, cannot instantly identify the class as abstract and has to reflect.

Reason and Rationale: We expect this smell to appear mainly because of carelessness mistakes.

Correction: The correction of this smell is trivial. The affected class just has to be declared as abstract.

Automatic Detection: Occurrences of this smell can only partly be detected automatically. When a concrete class has subclasses, it might be a true case of this smell [2]. If any of the subclasses are abstract, it is even more likely that there is an issue. However, manual evaluation is still required, as it might be the case that the superclass is validly concrete. In constellations, where all subclasses of the concrete abstract class are in external metamodels, the smell is not detectable if the external metamodels are not analyzed. This smell might lead to wrongful detections of the dead class smell

(5.7). This is the case when the class and its superclasses are never used within the metamodel but carry the information of an abstract concept that should be specialized in an external metamodel.

Occurrences: Within the PCM, the `CallReturnAction` class from the `SEFF` package is a true occurrence of the concrete abstract class smell. It is a concrete superclass and cannot be instantiated by the custom build graphical model editors of the PCM bench. This class cannot be meaningfully instantiated, as it or its superclasses have no container. However, it can still be confusing for a developer.

5.9 Duplicate Features in Sibling Classes

Description: In metamodels, classes represent concepts and inheritance is used to specialize concepts by providing a more comprehensive specification. For example, an abstract component type only describes that a component type has an interface. This class can be specialized into a component type that allows having internal components. In case a class has multiple children, of which some realize the same feature, this can be seen as a redundancy in the model.

In OO, one could argue that this issue falls under the duplicate code smell [9].

Consequence: Redundant declarations harm maintainability, as they must be maintained equally in all classes. If one class is overlooked, the metamodel degrades, which hinders long time evolution of the metamodel. They also have a negative impact on implementing transformations, as for each sibling class, the transformation rule must be able to support the feature. This is necessary, as from a syntactic viewpoint on the model these features are different.

Reason and Rationale: Duplicated features can appear when metamodels are altered iteratively. In that case, one of the sibling classes is extended with a specific feature, and later another sibling is extended in the same way. Through this process, more and more classes have a semantically identical feature, but they are declared syntactically as different features. While in some cases this situation may be the effect of limited time, carelessness, or overlooked, it can also be made intentional. The latter case occurs when not all siblings require the feature.

Correction: To mitigate the issue of duplicate features, in OO, a refactoring would move the feature up to the parent class in case all siblings have the feature [19]. However, this can be in violation with the underlying semantics of the concepts, which are expressed in the classes. Furthermore, the pull up cannot be used in cases where not all, but some siblings declare the same feature. An alternative strategy is to define an interface that provides the feature in question and inherit the interface by all siblings that require the particular feature. The strategy has two advantages. First, the meaning of the feature is encapsulated in its own concept. Second, the interface can be used in transformations. Therefore, the transformation must only test whether the interface exists instead of testing multiple classes.

Automatic Detection: An automatic detection of duplications based on name and type is unreliable, as the detection is based only on syntactic properties. Therefore, the detection may result false positives and false negatives. First, there could be identical typed and named features that do not represent the same relationship. Second, features may be named differently, but still represent the same idea. Therefore, manual intervention is required.

Occurrences: In the PCM, the classes `OperationSignature`, `InfrastructureSignature` and `EventType` have all a property `returnType` with the same intended semantic. These could be extracted into an `IValueReturning` interface, which is inherited by the three classes.

5.10 Classification by Enum

Description: If an enum is used as an alternate way to classify a class, we consider it a bad smell. This should not be confused with using an enum to model a mere property.

Using an enum for classification is one possible solution of how to model multiple orthogonal classifications. Part a) of Figure 7 illustrates the problem. It should be possible to classify the class Base as either A1 or A2 and additionally either as B1 or B2. This is not possible by just using an inheritance hierarchy of the depth of just one. Part b) of Figure 7 shows a solution by using an enum for the second classification dimension.

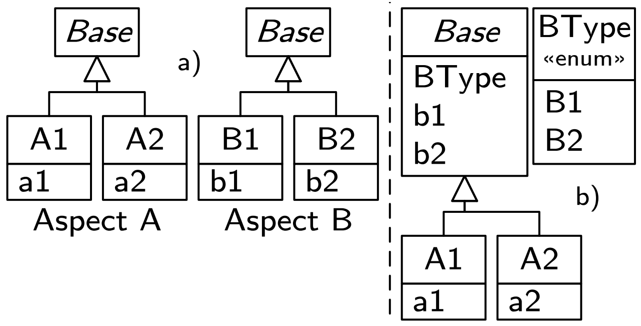

Figure 7: a) Problem of Orthogonal Classifications b) Classification by Enum

The naive solution to modeling orthogonal classifications is shown in Figure 8. There, every possible combination is explicitly modeled by inheritance. This obviously has several disadvantages. It produces high amounts of classes. Although, a single classification dimension is externally extensible, it is not possible to develop independent extensions, as every combination of every dimension has to be modeled.

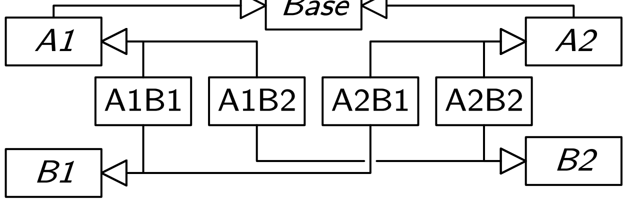

Figure 8: Naive Solution to Orthogonal Classifications

In OO, classification by enum is also a problematic solution to the problem of orthogonal classification dimensions.

However, we are not aware of any bad smell definition.

Consequence: Using an enum for additional classifications makes the classification impossible to extend externally. Further, in contrast to classification by inheritance, it is not possible to add features to parts of the classification selectively. This might lead to the developer adding features to the base class, which are only used for specific values of the enum. By doing that, the complexity of the class increases unnecessarily and its understandability suffers. This is shown in part `b)` of Figure 7.

Reason and Rationale: As already stated, using an enum for classification is one possible solution of how to model multiple orthogonal classifications. In most situations, however, it is not a very suited one. Developers use it because of lack of knowledge of more appropriate solutions. In addition, it looks simple and little intrusive compared to the naive approach (see Figure 8). If, however, the developer wants one classification to be closed for extension and it does not carry any new features that vary for its subtypes, classification by enum can be legitimately used.

Correction: There are two ways to resolve this smell. If the classification is already known when the metamodel is initially implemented, or it is possible to modify the metamodel, the *composition over inheritance principle* [10] should be applied. This is shown in part `a)` of Figure 9. If not, stereotypization should be applied. This is shown in part `b)` of Figure 9.

Automatic Detection: This smell is not automatically detectable. One could scan for each usage of an enum. However, not every usage is a classification. Therefore, each enum usage has to be checked manually.

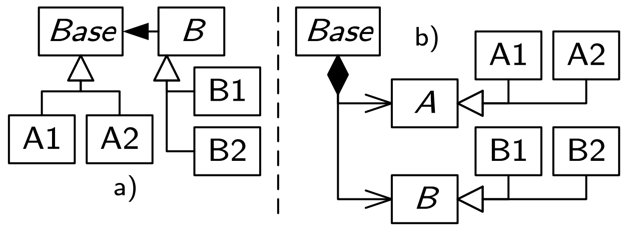

Figure 9: Solutions to Orthogonal Classifications

Occurrences: In the PCM, the classification by enum occurs in the class `ImplementationComponentType`. It contains an enum that declares its component type: business component or infrastructure component. This classification is orthogonal to the classification of atomic vs. composite, which is implemented by inheritance. This enum makes it impossible to add further component types without intrusively modifying the PCM.

6. CONCLUSION

Within this paper, we presented a list of metamodel smells we found in the current version of the PCM. We identified 10 different types of smells. Simple metamodel errors (which cause validation errors and prohibit code generation) are not included, as the PCM is already in operation and thus does not contain any.

Two of the presented smells are exclusive to metamodels. The remaining eight smells also represent issues in OO. However, there are differences in the usage of metamodeling, OO design and code. Therefore, in OO there is not as much emphasis on the smells which were presented here.

The smells presented in this paper are specific to languages that are similar to EMOF. All smells are concerned with the following basic concepts: classes, relations and attributes. Some smells are concerned with more specific meta-language features: explicit containments (5.1, 5.2), modularization (5.3, 5.4, 5.5), inheritance (5.8, 5.9) and enums (5.10).

Four of the smells might be detected by scanning for antipatterns. However, all of them still have to be reviewed, as there are circumstances where an occurrence is not necessarily bad design. The remaining six smells can only be detected by manual review.

For each smell, we explain how it might come into being. Some smells are built in by mere carelessness or lack of knowledge. Therefore, knowledge of these metamodel smells is very valuable for metamodel developers. Other smells, however, do only manifest with time, when multiple evolution steps have been performed (some of them in a short-sighted manner).

For each smell, we explained the effects we observed and further consequences that we expected. Some smells add unnecessary complexity. Others simply impair understandability by obfuscating design decisions or the intended structure of the metamodel. Some have negative effects on coupling and cohesion of packages. Thus, these metamodel smells have detrimental effects on metamodel maintainability. The consequences are even worse, if the metamodel is long living, is evolved and the smells accumulate without being fixed. Metamodels tend to live in metamodel-centric software systems. Many tools, like editors, analyzers and simulators, are built upon them. If the metamodel is changed, all tools have to be fixed. The effort caused by resolving smells in the metamodel increases over time, as new dependencies pile up. Thus, smells should be fixed as early as possible.

Future work includes inspecting further metamodels also including less mature ones. Further, results from smell and error detection tool could be incorporated and analyzed.

7. ACKNOWLEDGMENTS

This work was supported by the Helmholtz Association of German Research Centers and the DFG (German Research Foundation) under the Priority Program SPP1593: Design For Future – Managed Software Evolution. We would like to thank Anne Koziolek, Richard Rhinelander, Kiana Rostami, Dominik Werle, Kateryna Yurchenko and the anonymous peer-reviewers for their valuable input.

References

[1] T. Arendt and G. Taentzer. “A tool environment for quality assurance based on the Eclipse Modeling Framework”. In: *Automated Software Engineering* 20.2 (2013), pp. 141–184.

[2] T. Arendt et al. “Defining and checking model smells: A quality assurance task for models based on the

eclipse modeling framework". In: *BENEVOL workshop.* 2010.

[3] V. Basili et al. "A validation of object-oriented design metrics as quality indicators". In: *Software Engineering, IEEE Transactions on* 22.10 (Oct. 1996).

[4] P. Bourque et al. *Guide to the software engineering body of knowledge.* IEEE, 2014.

[5] S. R. Chidamber and C. F. Kemerer. "Towards a Metrics Suite for Object Oriented Design". In: *SIGPLAN Not.* 26.11 (Nov. 1991), pp. 197–211.

[6] J. Di Rocco et al. "Mining Metrics for Understanding Metamodel Characteristics". In: *Workshop on Modeling in Software Engineering.* ACM, 2014.

[7] M. Elaasar. "An approach to design pattern and anti-pattern detection in mof-based modeling languages". PhD thesis. Carleton University Ottawa, 2012.

[8] M. Elaasar et al. "Domain-Specific Model Verification with QVT". In: *ECMFA*. Springer, 2011.

[9] M. Fowler et al. *Refactoring: Improving the Design of Existing Code.* Addison-Wesley, 1999.

[10] E. Freeman et al. *Head First Design Patterns.* Head First. O'Reilly Media, 2004.

[11] I. García-Magariño et al. "An evaluation framework for MAS modeling languages based on metamodel metrics". In: *Agent-Oriented Software Engineering* (2009).

[12] M. Genero et al. "Building measure-based prediction models for UML class diagram maintainability". English. In: *Empirical Software Engineering* 12 (5 2007).

[13] J. J. C. Gómez et al. "Searching the Boundaries of a Modeling Space to Test Metamodels". In: *Software Testing, Verification, and Validation, 2008 International Conference on* (2012), pp. 131–140.

[14] K. Julisch. "Understanding and overcoming cyber security anti-patterns". In: *Computer Networks* 57.10 (2013), pp. 2206–2211.

[15] R. Jung et al. "A Method for Aspect-oriented Meta-Model Evolution". In: *Proceedings of the 2Nd Workshop on View-Based, Aspect-Oriented and Orthographic Software Modelling.* VAO '14. York, United Kingdom: ACM, July 2014, 19:19–19:22.

[16] R. Jung et al. "GECO: A Generator Composition Approach for Aspect-Oriented DSLs". In: *Theory and Practice of Model Transformations: 9th International Conference on Model Transformation, ICMT 2016.* Springer International Publishing, 2016, pp. 141–156.

[17] M. E. Kramer et al. "Extending the Palladio Component Model using Profiles and Stereotypes". In: *Palladio Days 2012.* Ed. by S. Becker et al. Karlsruhe Reports in Informatics ; 2012,21. Karlsruhe: KIT, Faculty of Informatics, 2012, pp. 7–15.

[18] P. Langer et al. "EMF Profiles: A Lightweight Extension Approach for EMF Models". In: *Journal of Object Technology* 11.1 (2012), 8:1–29.

[19] K. Lano and S. K. Rahimi. "Case study: Class diagram restructuring". In: *Proceedings Sixth Transformation Tool Contest, TTC 2013, Budapest, Hungary, 19-20 June, 2013.* 2013, pp. 8–15.

[20] J. J. López-Fernández et al. "Assessing the Quality of Meta-models". In: *Proceedings of the 11th Workshop on Model Driven Engineering, Verification and Validation (MoDeVVa).* 2014, p. 3.

[21] M. Manso et al. "No-redundant Metrics for UML Class Diagram Structural Complexity". In: *Advanced Information Systems Engineering.* Vol. 2681. Lecture Notes in Computer Science. Springer Berlin Heidelberg, 2003, pp. 127–142.

[22] M. Marchesi. "OOA metrics for the Unified Modeling Language". In: *Proceedings of the Second Euromicro Conference on Software Maintenance and Reengineering.* Mar. 1998, pp. 67–73.

[23] R. Martin. *Agile Software Development: Principles, Patterns, and Practices.* PH, 2003.

[24] Object Management Group (OMG). *MOF 2.4.2 Core Specification (formal/2014-04-03).* 2014.

[25] Object Management Group (OMG). *Object Constraint Language, v2.0 (formal/06-05-01).* 2006.

[26] R. H. Reussner et al. *Modeling and Simulating Software Architectures – The Palladio Approach.* to appear. Cambridge, MA: MIT Press, Oct. 2016. 408 pp.

[27] L. Rising. *The Patterns Handbook: Techniques, Strategies, and Applications.* SIGS, 1998.

[28] C. U. Smith and L. G. Williams. "Software performance antipatterns". In: *Workshop on Software and Performance.* 2000, pp. 127–136.

[29] D. Steinberg et al. *EMF: Eclipse Modeling Framework.* second revised. Eclipse series. Addison-Wesley Longman, Amsterdam, Dec. 2008.

[30] M. Strittmatter and R. Heinrich. "A Reference Structure for Metamodels of Quality-Aware Domain-Specific Languages". In: *13th Working IEEE/IFIP Conference on Software Architecture.* Apr. 2016, pp. 268–269.

[31] M. Strittmatter and R. Heinrich. "Challenges in the Evolution of Metamodels". In: *3rd Collaborative Workshop on Evolution and Maintenance of Long-Living Software Systems.* Vol. 36. Softwaretechnik-Trends 1. 2016, pp. 12–15.

[32] M. Strittmatter and M. Langhammer. "Identifying Semantically Cohesive Modules within the Palladio Meta-Model". In: *Symposium on Software Performance.* Universitätsbibliothek Stuttgart, Nov. 2014, pp. 160–176.

[33] M. Strittmatter et al. "A Modular Reference Structure for Component-based Architecture Description Languages". In: *Model-Driven Engineering for Component-Based Systems.* CEUR, 2015, pp. 36–41.

[34] R. Subramanyam and M. Krishnan. "Empirical analysis of CK metrics for object-oriented design complexity: implications for software defects". In: *IEEE Transactions on Software Engineering* 29.4 (2003).

[35] E. Vépa et al. "Measuring model repositories". In: *Proceedings of the 1st Workshop on Model Size Metrics.* 2006.

[36] E. Yourdon and L. L. Constantine. *Structured Design: Fundamentals of a Discipline of Computer Program and Systems Design.* 1st. Prentice-Hall, 1979.